Hes·so VALAIS WALLIS

Haute Ecole de Gestion
Hochschule für Wirtschaft

Travail de Bachelor

Informatique de gestion

Architecture générique d'un serveur gRPC/REST en Golang

Auteur :

David Crittin

Professeur :

Jean-Luc Beuchat

Déposé à la HES-SO Valais-Wallis à Sierre, le 12 août 2022

#48b977

Résumé

L'*Applied Cryptography Laboratory* de l'Institut d'Informatique de Gestion a choisi le langage Go pour la majorité de ses développements. Le besoin a été émis d'avoir à disposition un serveur construit autour d'une architecture générique qui puisse être réutilisé dans les différents projets. Il devait être également possible d'utiliser des *middlewares* spécifiques en fonction des besoins.

Afin de réaliser la tâche demandée, nous avons procédé en plusieurs étapes.

Premièrement, nous avons pris en main le langage de développement Go, ainsi que l'écosystème gRPC. Différents tutoriels ont été suivis afin de se former et de monter en compétence.

Dans un second temps, la couche REST a été intégrée, également via le développement d'un serveur basique avec le plugin `grpc-gateway`.

Par la suite, le développement des *middlewares* demandant l'utilisation des *interceptors*, l'application *Hello World* d'auto-formation a été enrichie pour intégrer cet aspect et comprendre leur fonctionnement.

Sur ces bases, un serveur propre a été mis en place comme socle pour la suite des développements.

Ensuite, différents *middlewares* correspondants au cahier des charges ont été développés :

— gestion du *throttling*,

— chiffrement des communications,

— authentification avec Azure et Google,

— gestion des sessions.

La méthodologie a été la même pour chaque élément : réalisation d'un état de l'art du sujet à intégrer, recherche d'algorithmes ainsi que de bibliothèques *open-source* existantes, sélection d'une solution puis intégration dans le projet.

Finalement, une *Application Programming Interface* (API) de chiffrement externe a été intégrée et le serveur a été mis à disposition au travers d'une image Docker. Celle-ci est fonctionnelle pour utiliser la *client-side encryption* au sein de Google Workspace.

Mots clés : go, grpc, protocol buffer, grpc-gateway, REST, client-side encryption, Docker

Remerciements

Tout d'abord, je tiens à remercier Jean-Luc Beuchat pour ses conseils avisés ainsi que sa disponibilité tout au long de la réalisation de ce travail.

Ensuite, je souhaite particulièrement remercier ma compagne Emily qui m'aura soutenu non seulement durant la réalisation de ce travail de bachelor, mais durant l'ensemble de la formation.

Je tiens à relever la collaboration avec Nagib Aouini de la société DuoKey, qui a mis à disposition non seulement son temps mais aussi les ressources de son entreprise pour permettre la réalisation de ce travail.

Finalement, je remercie les relecteurs de ce document qui auront participé à l'effort visant à rendre un travail de qualité.

Table des matières

Table des figures

Table des figures

Liste des tableaux

Liste des abréviations

AEAD Authenticated Encryption with Associated Data. vii, 40–43, 45, 46

API Application Programming Interface. ii, 4, 7, 9–13, 17, 29, 39, 43, 44, 55, *glossaire :* Application Programming Interface

CA Certification Authority. 26, 50

CORS Cross-Origin Ressource Sharing. 46, *glossaire :* Cross-Origin Ressource Sharing

DDoS Distributed Denial-of-Service. 17

DEK Data Encryption Key. 40, 41, 43, 45, *glossaire :* Data Encryption Key

gRPC Google Remote Procedure Call. ii, 1, 5, 6, 8, 10, 11, 16, 50

HMAC Hash-based message authentication code. 35

HTTP Hyper Text Transfer Protocol. 14, 64

IdP Identity Provider. 40, 41, 46, *glossaire :* identity provider

JSON JavaScript Object Notation. 34, 40, 41, 44

JWKS JSON Web Key Sets. 31, 50

JWT JSON Web Token. 7, 29, 30, 33–35, 40, 41, 54

KACLS Key Access Control List Service. 40, 41, 43, 46, 52, *glossaire :* Key Access Control List Service

KMS Key Management Service. 40, 52, *glossaire :* Key Management Service

PoC Proof of Concept. 43, 50, *glossaire :* proof of concept

REST Representational State Transfer. ii, 1, 11

RFC Request For Comments. 23, 34

RPC Remote Procedure Call. 5–7, 44

SHA Secure Hash Algorithms. 35

SSL Secure Sockets Layer. 23, 25

TLS Transport Layer Security. 23–26, 51

URL Uniform Resource Locator. 29, 41

UTC Universal Time Coordinated. 64

YAML Yet Another Markup Language. 49, *glossaire :* YAML

1 Introduction

1.1 Contexte

Ce travail de bachelor est réalisé dans le cadre de la formation en Informatique de Gestion de la HES-SO Valais Wallis. Le sujet a été proposé par Jean-Luc Beuchat pour l'*Applied Cryptography Laboratory*. Il était nécessaire d'avoir à disposition un serveur gRPC/REST à l'architecture générique modulable à l'aide de différents *middlewares*, selon les besoins du projet.

1.2 Objectifs

Le but de ce travail est de fournir un serveur gRPC/REST à l'*Applied Cryptography Laboratory* de l'Institut d'Informatique de Gestion. Go est un des langages de programmation de prédilection du laboratoire.

L'architecture du serveur proposé doit permettre l'utilisation de *middlewares*, qui permettent d'activer ou non des fonctionnalités précises. Le mandant en a transmis plusieurs à développer via le cahier des charges de ce travail de bachelor, tels que le *throttling*, ou encore la gestion des sessions.

Il pourra ensuite être intégré dans un écosystème existant et intégrer une API de chiffrement externe telle que Google *Client Side Encryption*. L'ensemble sera ensuite mis à disposition via une image Docker.

1.3 Méthodologie

Un *product backlog* a été défini lors du démarrage de ce travail, afin d'identifier les tâches principales à réaliser, ainsi que les sous-tâches qui en découlent. Nous avons utilisé la version *cloud* d'Atlassian JIRA [1] qui nous a permis de créer des *epics*, *stories* et *tasks*.

1.4 Planification

Ce travail a été réalisé du 24 février au 12 août 2022, durant le dernier semestre de formation à la HES-SO Valais Wallis. Les actions à réaliser ont été définies en plusieurs *epics* qui ont été réalisées les unes après les autres, selon le planning qui est disponible en annexe IV.

1. https://tb-grpc-hevs.atlassian.net/jira

1.5 Environnement de développement

Dans le cadre de ce travail, une machine tournant sous Windows 11 a été utilisée, avec le *Windows Subsystem for Linux (WSL)*. Pour le développement, la version 1.65.2 de *Visual Studio Code* a été employée.

Le code produit est disponible dans le groupe Gitlab `HEVs Travail de bachelor gRPC-REST`[2]. Il est composé de différents projets :

— `thesis` contient le code LaTeX du rapport,

— `go-grpc-helloworld` contient le code du projet de prise en main,

— `go-grpc-rest-server` contient le code du serveur générique à destination de l'*Applied Cryptography Laboratory*.

2. https://gitlab.com/hevs-travail-de-bachelor-grpc-rest

2 Présentation des concepts et technologies

2.1 Langage Go

Go est un langage de programmation compilé développé par Google depuis 2009, sa version 1.0 a été publiée en 2012. Au moment de la rédaction de ce document [1], il est dans sa version majeure 1.18.

FIGURE 2.1 – *Logo officiel de Go*
Source: *de l'auteur à partir de https://go.dev/blog/go-brand*

Le langage Go présente les caractérisques suivantes :

— il est *open-source* et son code est disponible sur GitHub [2]. Chacun est ainsi libre de télécharger le code source, de le modifier et de contribuer à son développement.

— c'est un langage avec typage statique, c'est à dire qu'il est nécessaire de déclarer le type de données représenté dans la variable. Une variable de type `string` sera ainsi déclarée comme suit :

```
myString string = "hello"
```

Il est également possible d'utiliser le *type inference*, où le compilateur va détecter le type de la variable, à l'aide du mot-clé `var`.

— la concurrence est supportée à l'aide des *goroutines*. Une méthode peut être appelée sans bloquer l'exécution du reste du programme à l'aide du mot-clé `go`. Contrairement à un *thread* qui est géré par le système d'exploitation, une *goroutine* le sera par le *runtime* go.

— de nombreux outils sont mis à disposition dans la bibliothèque standard [3], tels que `go fmt` qui permet de formater le code selon le standard Go, ou encore `go get` qui permet de télécharger des bibliothèques depuis un *repository* GitHub et de les importer aisément dans un projet.

1. au 9 août 2022
2. https://github.com/golang/go
3. https://pkg.go.dev/std

— la mise en place d'une suite de tests unitaires est possible de manière automatisée [4]. Les fichiers terminant par `_test.go` seront pris en compte lors de l'exécution de la commande `go test`. Il est possible également d'obtenir la couverture de code de manière native [5].

Voici un exemple de *Hello world* en Go :

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          fmt.Printf("Hello, world")
7  }
```

De nombreuses entreprises ont adopté ce langage pour leurs développements, par exemple :

— Google, entreprise active dans de nombreux secteurs technologiques, utilise Go dans plusieurs projets tels que les indexeurs Google Search, ainsi que pour son service Firebase Hosting, [6] ;

— bitly, un service d'*url shortening*, a choisi Go comme langage principal [7] ;

— Dropbox, fournisseur de stockage cloud, a migré des éléments de son *backend* en Go [8] ;

— HashiCorp, société développant des outils utiles aux infrastructures *cloud*, propose le produit Vault [9] qui est écrit en Go [10]. C'est un outil permettant le stockage des *tokens*, mots de passe, certificats, clés d'API de manière sécurisée.

2.2 Protocol buffer

Protocol buffers, ou *protobuf*, est un format de données *cross-platform* développé par Google. Son but est de faciliter l'échange de données sérialisées, et ce de manière indépendante de la plateforme ou du langage de programmation utilisé. Il est par exemple possible de définir des données à échanger entre un serveur écrit en Go et un client en Java. Le code est généré pour chaque langage sur la base du contenu du fichier `.proto`.

4. https://pkg.go.dev/testing
5. https://go.dev/blog/cover
6. https://go.dev/solutions/google/
7. https://bitly.com/blog/why-we-write-everything-in-go/
8. https://dropbox.tech/infrastructure/open-sourcing-our-go-libraries
9. https://www.vaultproject.io/
10. https://github.com/hashicorp/vault

Dans sa version 2.0, la génération de code est possible en C++, Java, C# et Python. Actuellement en version majeure 3.0 depuis 2016 (avec des *releases* mineures, la dernière en date étant la 3.21 [11]), il est possible d'avoir du code généré en Go, Ruby, Objective-C ainsi que Javascript. Le code de *protobuf* est *open-source* [12] et disponible sur GitHub [13].

La structure des données à échanger est définie dans un fichier `.proto`, puis il est compilé à l'aide de l'utilitaire `protoc`. Le code généré dans un fichier `.pb.go` peut ensuite être appelé dans une application Go.

Il est possible d'agrémenter le code avec des imports, comme dans notre projet, en ajoutant des annotations pour Swagger par exemple.

```
1  syntax = "proto3";
2
3  package v1;
4  option go_package = "./v1"; // import path utilisé pour la génération du
   ↪  code
5
6  message TokenRequest {
7  }
8
9  message TokenResponse {
10     string token = 1;
11 }
```

2.3 gRPC & gRPC-Gateway

gRPC est un *framework* pour l'utilisation de RPC via `HTTP/2`, publié par Google en 2016 dans sa première version.

Une application cliente peut faire appel à une méthode exposée sur un serveur d'application. L'utilisation de gRPC va de paire avec l'utilisation des *protocol buffers*.

11. au 9 août 2022
12. https://developers.google.com/protocol-buffers/docs/overview#protocol_buffers_open_source_philosophy
13. https://github.com/protocolbuffers/protobuf

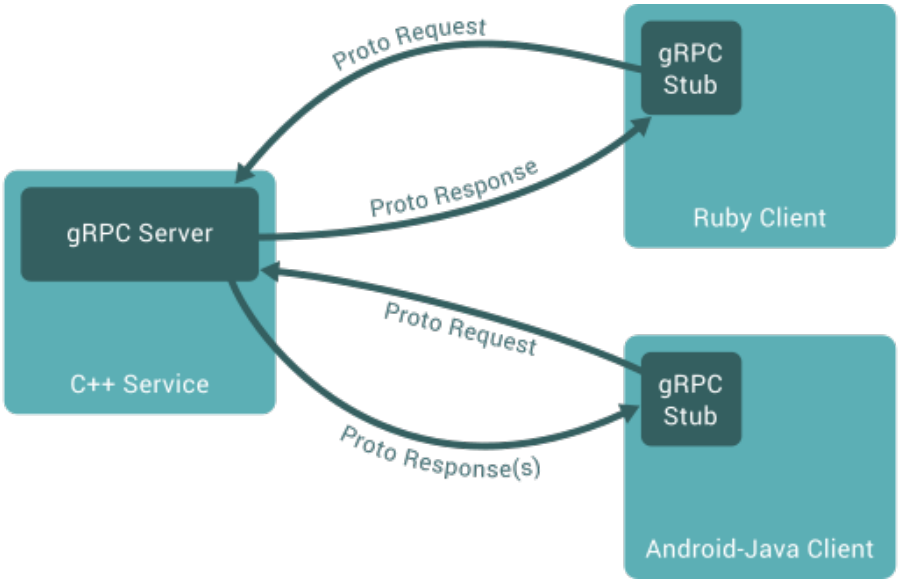

FIGURE 2.2 – *Schéma d'utilisation gRPC et protobuf*
Source: *de l'auteur à partir de https://grpc.io/docs/what-is-grpc/introduction/*

gRPC est utilisée entre autres par la société de service de *streaming* Netflix pour son *load-balancing*[14].

Dans ce projet, gRPC-Gateway[15] sera également utilisé. Il s'agit d'un *plugin* de l'utilitaire `protoc`. Cela nous permet d'avoir une interface RPC exposée pour appeler le service gRPC.

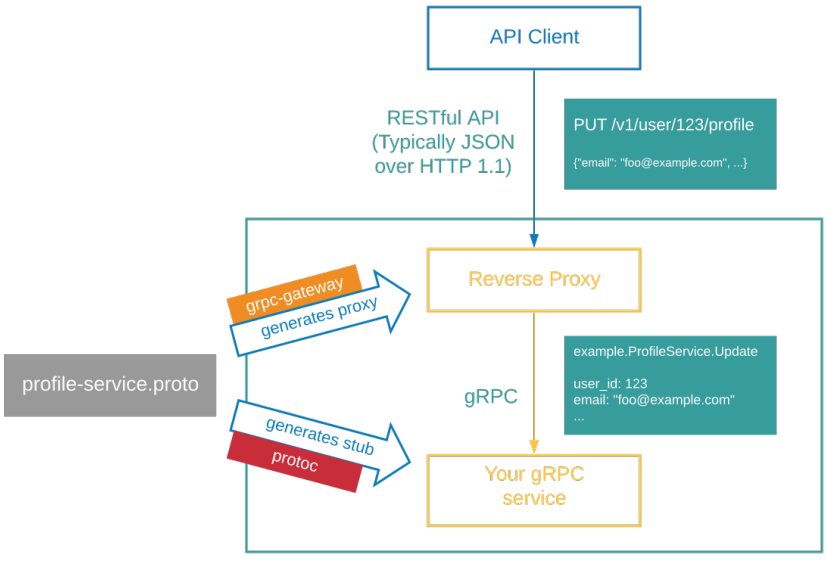

FIGURE 2.3 – *Schéma d'utilisation gRPC et gRPC-Gateway*
Source: *de l'auteur à partir de https://grpc-ecosystem.github.io/grpc-gateway/*

14. https://github.com/Netflix/ribbon
15. https://grpc-ecosystem.github.io/grpc-gateway/

2.4 Swagger

Swagger est une suite d'outils pour le développement d'API. Afin d'interagir avec celle qui a été développée, nous pouvons générer avec `protoc` un fichier `swagger.json`, qui correspondra à la définition *OpenAPI*[16] de l'API. Ce fichier sera ensuite exposé au travers de Swagger UI qui permettra de tester les *endpoints* exposés par les services RPC.

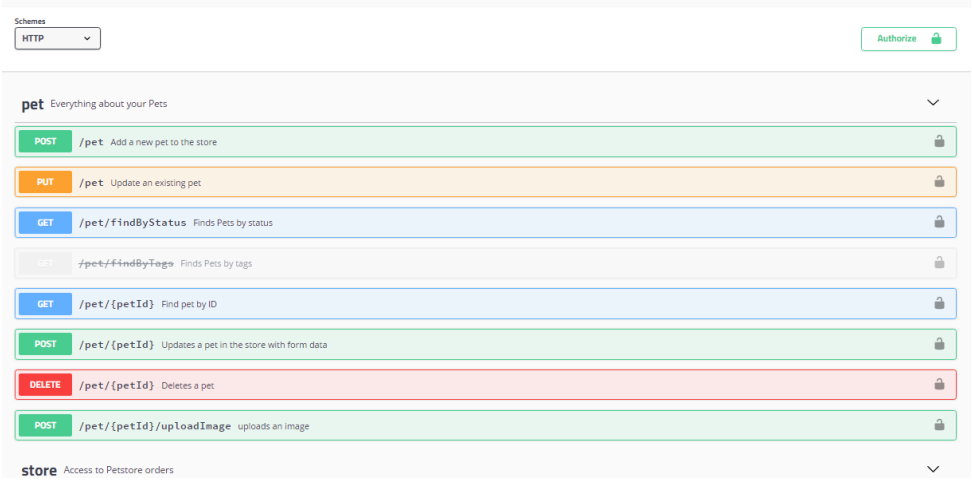

FIGURE 2.4 – *Exemple de routes dans Swagger UI*
Source: *de l'auteur à partir de https:// swagger.io/ tools/ swagger-ui/*

Il est également possible d'activer la gestion de l'authentification *bearer*[17] qui permet de passer un *token* JWT aux routes exposées. L'information sera ensuite passée via les *headers* lors de l'envoi de la requête.

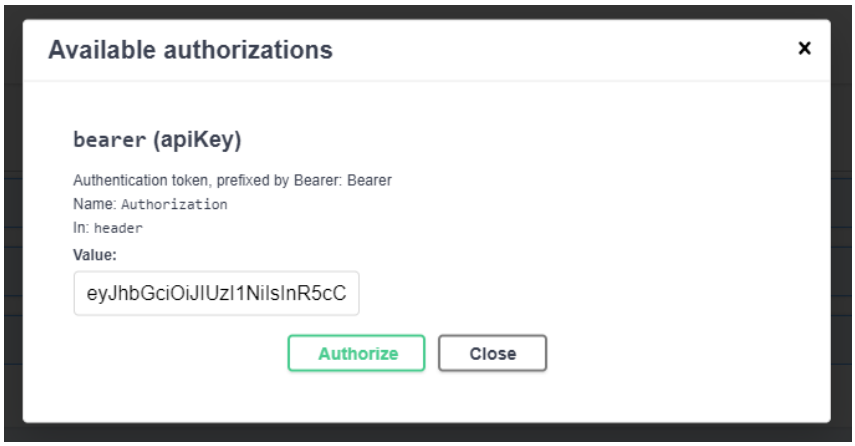

FIGURE 2.5 – *Paramétrage d'un token Authorization*
Source: *de l'auteur*

16. https://swagger.io/specification/
17. https://swagger.io/docs/specification/authentication/bearer-authentication/

3 Prise en main de l'environnement Golang et gRPC/REST

3.1 Prérequis

3.1.1 Installation de Go

Pour pouvoir commencer le développement en Go, il est nécessaire de télécharger et d'installer le compilateur à partir du site officiel [1]. La version utilisée dans le cadre de ce travail est la version 1.17.6.

3.1.2 Installation de Protocol Buffer

Dans le cadre de ce travail, nous allons utiliser Protocol Buffer de Google qui permet la définition de service et la sérialisation de données. Il est nécessaire d'installer le compilateur `protoc`. La marche à suivre, selon le site officiel [2], est la suivante :

```
apt install -y protobuf-compiler
protoc --version  # Ensure compiler version is 3+
```

Sur la machine de développement utlisée, le résultat de la commande `protoc -version` est le suivant :

```
libprotoc 3.6.1
```

3.1.3 Installation des *plugins* gRPC

Selon la marche à suivre du site officiel gRPC [3], il est nécessaire d'installer les *plugins* suivants `protoc-gen-go` et `prtoc-gen-go-grpc`. Ceci se fait à l'aide des commandes :

```
go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.26
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.1
```

Il faut ensuite exporter la variable d'environnement `PATH`

```
export PATH="$PATH:$(go env GOPATH)/bin"
```

1. https://go.dev/doc/install
2. https://grpc.io/docs/protoc-installation
3. https://grpc.io/docs/languages/go/quickstart/

3.1.4 Installation de `buf`

Afin de faciliter la génération de code à partir des fichiers `.proto`, l'outil `buf`[4] sera utilisé. Pour l'installer, un exécutable est disponible sur le site officiel[5]. La version utilisée dans ce travail est la 1.1.0[6].

Nous pourrons ensuite configurer les fichiers `.yaml` nécessaires, à savoir `buf.yaml` et `buf.gen.yaml`. Le contenu ci-dessous représente le premier fichier, où la version ainsi les dépendances nécessaires sont configurées.

```yaml
version: v1
name: buf.build/bachelor/go-grpc-helloworld
deps:
  - buf.build/googleapis/googleapis
  - buf.build/grpc-ecosystem/grpc-gateway
```

Dans le second fichier, nous pouvons indiquer quelles opérations effectuer lors de la génération des fichiers à l'aide de la commande `buf generate`. Nous indiquons ici de générer les fichiers nécessaires pour Go, go-grpc et grpc-gateway. Le dernier *plugin* permet de générer la définition de l'API pour Swagger.

```yaml
version: v1
plugins:
  - name: go
    out: ./
    opt:
      - paths=source_relative
  - name: go-grpc
    out: ./
    opt:
      - paths=source_relative
  - name: grpc-gateway
    out: ./
    opt:
      - paths=source_relative
  - name: openapiv2
    out: ../../swagger/v1
```

4. https://github.com/bufbuild/buf
5. https://docs.buf.build/installation#windows-support
6. résultat de la commande `buf -version`

3.2 Serveur gRPC/REST simple documenté avec Swagger

Dans ce chapitre, nous allons présenter le projet avec lequel nous avons pu monter en compétence et prendre en main les différents aspects de l'écosystème gRPC. L'ensemble du code présenté est disponible dans le *repository* Gitlab `hevs-travail-de-bachelor-grpc-rest/go-grpc-helloworld`.

3.2.1 Structure du projet

Le projet est organisé selon la hiérarchie de répertoires suivante :

```
go-grpc-helloworld
├── cmd
│   └── go-grpc-helloworld
│       ├── client
│       │   └── client.go
│       └── server
│           └── server.go
└── internal
    ├── api
    │   ├── proto
    │   │   ├── third_parties
    │   │   └── v1
    │   │       ├── hello_world
    │   │       │   └── hello_world.proto
    │   │       ├── buf.gen.yaml
    │   │       └── buf.yaml
    │   └── swagger
    ├── client
    │   └── client.go
    └── server
        └── server.go
```

Comme aucun standard officiel n'est défini, cette structure est inspirée de ce qui est proposé par la communauté [7].

Le premier dossier `cmd` regroupe les fichiers `.go` qui sont nécessaires à l'exécution du serveur et du client. L'application peut ensuit être exécutée à l'aide de la commande `go run cmd/go-grpc-helloworld/server/server.go`.

Le dossier `internal` regroupe l'ensemble du code de l'application. Il est composé de plusieurs sous-répertoires : `api`, `client` et `server`. Le dossier `api` contient les différents fichiers nécessaires à notre API, avec les bibliothèques externes dans le dossier `third_parties`, et

7. https://github.com/golang-standards/project-layout

les fichiers *protobuf* correspondant à notre service décrit en Protocol Buffer dans le dossier `proto`. Ce répertoire contient également des fichiers `.yaml` nécessaires à la génération du code à l'aide de l'utilitaire `buf`.

À la racine du dossier `api`, nous retrouvons également le dossier `swagger`. Il contient deux dossiers : `swagger-ui` et `v1`. Le premier répertoire contient les fichiers nécessaires au fonctionnement de Swagger UI, et le second va contenir le fichier de définition de notre API `swagger.json` qui est généré automatiquement lors de la construction de notre code à partir des fichiers `.proto`.

Finalement, deux dossiers `server` et `client` contiennent le code applicatif du serveur et du client. C'est là que se trouve ce qui est nécessaire au bon fonctionnement de l'application.

3.2.2 Code de notre application

Pour prendre en main le langage Go, nous avons développé une application basique avec gRPC/REST. La structure de cette application a été présentée dans la section précédente, nous allons ici nous intéresser au code.

Nous allons commencer par présenter le contenu du fichier *protobuf*, à savoir notre fichier `hello_world.proto`.

```
syntax = "proto3";
package v1;
option go_package = "./v1";

import "google/api/annotations.proto";
import "protoc-gen-openapiv2/options/annotations.proto";

option (grpc.gateway.protoc_gen_openapiv2.options.openapiv2_swagger) = {
        info: {
                title: "TB gRPC - Hello World Swagger";
                version: "0.1";
                contact: {
                        name: "David Crittin";
                        email: "david.crittin@students.hevs.ch";
                };
        }
};
service HelloWorld {
    rpc SayHello(HelloRequest) returns (HelloReply) {
        option (google.api.http) = {
            post: "/v1/sayhello"
            body: "*"
          };
    }
```

```
}

message HelloRequest {
    string name = 1;
}

message HelloReply {
    string greetings = 1;
}
```

Le fichier est composé de trois blocs principaux. Le premier bloc permet de définir quelques options spécifiques qui se retrouveront dans le fichier de définition de l'API à destination de Swagger.

Le second représente la définition du service et ses méthodes disponibles, ici `HelloWorld` avec une méthode `SayHello`. Elle utilise deux *messages* qui sont définis plus bas, `HelloRequest` et `HelloReply`. Ces derniers définissent leurs paramètres respectifs, `name` pour le premier et `greetings` pour le second. Une fois les fichiers générés à l'aide de la commande `buf generate`, ils peuvent être utilisés dans notre application.

Le code qui suit est celui du fichier `server.go`. L'intégralité de celui-ci étant disponible dans le Gitlab du projet, seul quelques portions du codes vont être présentées. Tout d'abord, les imports nécessaires, où se trouvent les différentes bibliothèques utilisées et le code du *protobuf* généré.

```
import (
        pb "bachelor/go-grpc-helloworld/internal/api/proto/v1/hello_world"
        "context"
        "log"
        "net"
        "net/http"
        "os"

        "github.com/grpc-ecosystem/grpc-gateway/runtime"
        "google.golang.org/grpc"
        "google.golang.org/grpc/credentials/insecure"
)
```

Ensuite, la méthode `SayHello` pour laquelle le comportement souhaité a été implémenté. Dans le cadre du projet de formation, celle-ci renverra une salutation telle que "*Hello David*".

```
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest)
↪  (*pb.HelloReply, error) {
        log.Print("Got a SayHello request with value " + in.Name)
        return &pb.HelloReply{Greetings: "Hello " + in.Name + "!"}, nil
```

```
}
```

Lors du démarrage du serveur, le service est enregistré auprès de ce dernier.

```
// On crée un nouveau serveur gRPC
s := grpc.NewServer()
// On enregistre le service auprès du serveur
pb.RegisterHelloWorldServer(s, &server{})
```

Ensuite, les connexions nécessaires à *grpc-gateway* ainsi qu'à Swagger sont créées. Le code généré à partir des *protobuf* va également mettre à disposition une méthode permettant d'enregistrer un *handler* de notre service auprès de la *gateway*.

```
// On crée une connexion client pour le serveur qui sera utilisée par
↪ grpc-gateway
conn, _ := grpc.DialContext(
    context.Background(),
    serverIp+":"+grpcPort,
    grpc.WithBlock(),
    grpc.WithTransportCredentials(insecure.NewCredentials()),
)

// Gateway
gwmux := runtime.NewServeMux()
// Swagger
muxSwagger := http.NewServeMux()
// Le muxSwagger englobe la gateway par défaut
muxSwagger.Handle("/", gwmux) // les routes par défaut sont gérées par le
↪ gwmux

// On enregistre le handler de notre service auprès de la gateway
pb.RegisterHelloWorldHandler(context.Background(), gwmux, conn)
```

La logique ci-dessous permet d'activer ou non les écrans Swagger UI en fonction de la présence du fichier de définition de l'API.

```
// Activation de swagger si on trouve le swagger.json
if _, err := os.Stat(swaggerJson); err == nil {
    log.Print("Swagger configuration found")
    muxSwagger.HandleFunc("/swagger.json", func(w http.ResponseWriter, r
↪ *http.Request) {
        // On indique au serveur de renvoyer le contenu du swagger.json
↪ quand le fichier est appelé
        http.ServeFile(w, r, swaggerJson)
    })

```

```
    // On indique notre racine swagger-ui
    fs := http.FileServer(http.Dir("internal/api/swagger/swagger-ui"))

    // Quand on appelle l'url "swagger-ui", on affiche le contenu du
    ↪ dossier
    muxSwagger.Handle("/swagger-ui/", http.StripPrefix("/swagger-ui", fs))
}
```

Finalement le serveur HTTP est créé et démarré.

```
// Création du serveur http
gwServer := &http.Server{
    Addr:    ":" + grpcGwPort,
    Handler: muxSwagger,
}

log.Print("Serving gRPC-Gateway on http://" + serverIp + ":" + grpcGwPort)
// On démarre le serveur et on affiche l'output des requests dans la
↪ console
log.Fatalln(gwServer.ListenAndServe())
```

Pour interagir avec le serveur, nous avons développé un client basique qui envoie une requête au service `SayHello` et afficher le résultat.

```
// Options pour la connexion au serveur
var opts []grpc.DialOption
// On autorise les connexions pas sécurisées
opts = append(opts,
↪ grpc.WithTransportCredentials(insecure.NewCredentials()))
// On crée une connexion
conn, err := grpc.Dial(*serverAddr, opts...)
// En cas d'erreur
if err != nil {
    log.Fatalf("fail to dial: %v", err)
}
// On ferme la connexion quand les méthodes environnantes seront
↪ terminées ("defer")
defer conn.Close()
log.Print("Connecting to gRPC server")

// On utilise la connexion créée précédemment
client := pb.NewHelloWorldClient(conn)
// On appelle le service SayHello avec une HelloRequest
reply, err := client.SayHello(context.Background(),
↪ &pb.HelloRequest{Name: "David"})
// En cas d'erreur
if err != nil {
```

```
21      log.Fatalf(err.Error())
22  }
23  // On affiche la réponse du serveur
24  log.Printf("Hello response from server %s", reply.Greetings)
```

4 | Middlewares

4.1 Introduction

Dans une application gRPC, les *middlewares* sont implémentés à l'aide d'*interceptors*. Ils permettent d'interagir avec un message *protobuf* lors de sa réception ou de son envoi. Cela permet par exemple d'ajouter ou de modifier une information reçue, d'intercepter un message à destination du serveur et d'effectuer des contrôles sur ce dernier.

Il y a deux types d'*interceptors*, qui comprennent un équivalent client et serveur. Il existe les *unary interceptors* et les *stream interceptors*. Leur différence principale consiste dans le type d'échange de données qui est effectué entre le client et le serveur. Dans le cadre d'un *unary interceptor*, il s'agit d'une simple requête. Pour les *stream interceptors*, ils agissent lors d'un *streaming* de données, et l'*interceptor* agit à chaque *chunk* échangé.

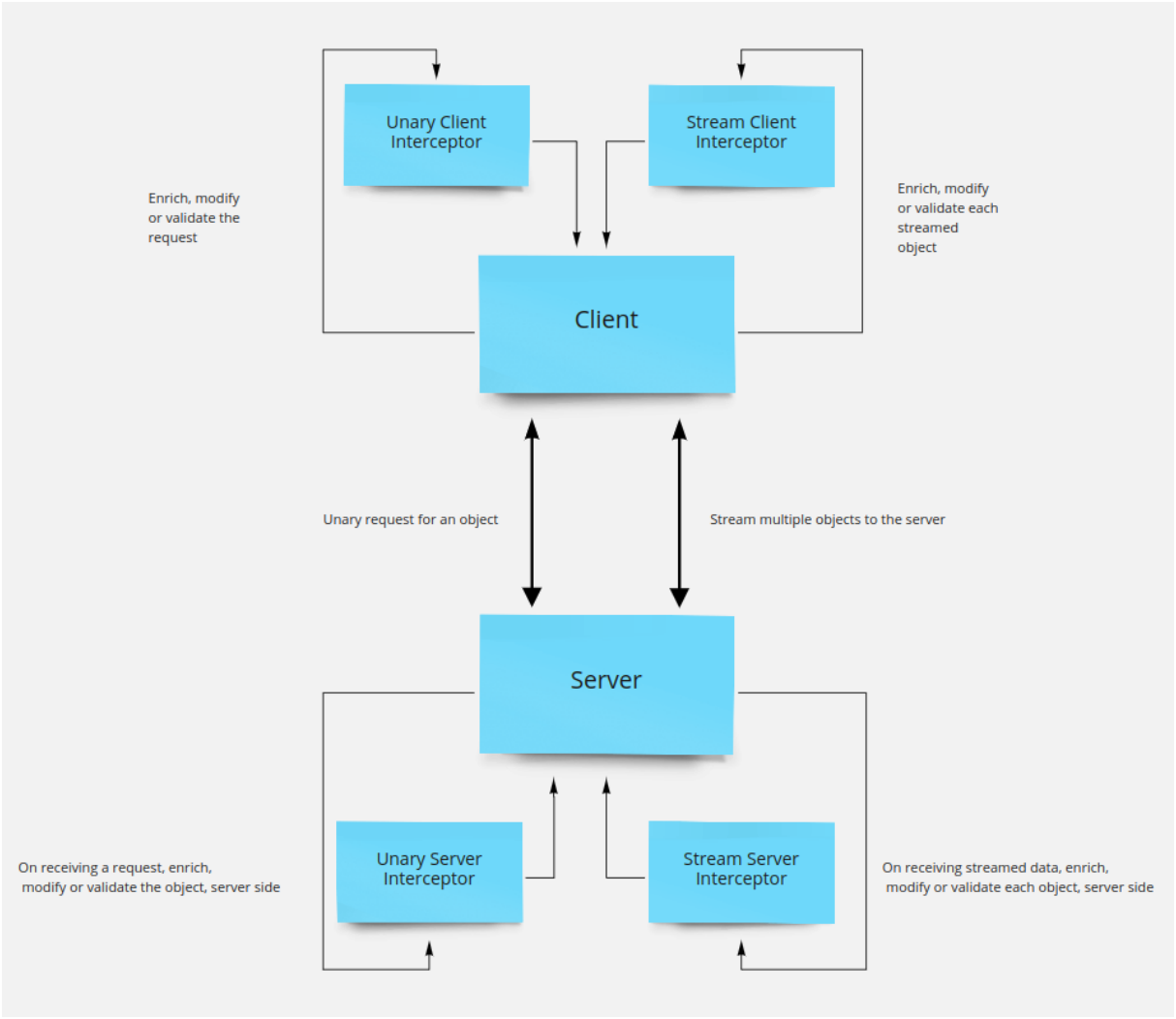

FIGURE 4.1 – *Types d'interceptors*
Source: *https:// edgehog.blog/ a-guide-to-grpc-and-interceptor-265c306d3773*

4.2 Méthodologie

Différents *middlewares* ont été proposés pour développement dans le cadre de ce travail. Pour chacun des sujets, la même méthodologie a été appliquée. Dans un premier temps, un état de l'art a été effectué. Ensuite, une comparaison des algorithmes ainsi que des éventuelles bibliothèques *open-source* existantes a été faite. Finalement, une solution a été sélectionnée puis implémentée dans le travail.

Concernant la comparaison des différentes bibliothèques, les paramètres suivants ont été observés :

Pour les comparer, nous nous sommes appuyés sur les critères suivants :

— documentation à disposition, exemples à disposition ;

— simplicité de mise en place ;

— algorithme(s) utilisé(s) (si relevant pour le *middleware* concerné) ;

— utilisation de bibliothèques externes (contenu du `go.mod`), afin de minimiser la quantité de code importée et ainsi diminuer le risque de *supply-chain attack*, à savoir une attaque via des dépendances compromises [1] ;

— date du dernier commit, relevée au 9 août 2022.

4.3 Throttling

4.3.1 État de l'art

Le *throttling*, étranglement en français, ou *rate limiting*, va permettre de protéger un service ou une API en limitant le nombre de requêtes auxquelles celui-ci va répondre, ou en limitant le nombre d'appels qu'un utilisateur va pouvoir effectuer dans une fenêtre temporelle. C'est une stratégie employée notamment pour mitiger les attaques DDoS.

Selon les différentes ressources [2] [3] [4] consultées, les algorithmes suivants sont principalement utilisés :

— *leaky bucket*,

— *token bucket*,

— *fixed window*,

— *sliding window*.

1. https://go.dev/blog/supply-chain
2. https://cloud.google.com/architecture/rate-limiting-strategies-techniques
3. https://blog.cloudflare.com/counting-things-a-lot-of-different-things/
4. https://www.quinbay.com/blog/understanding-rate-limiting-algorithms

4.3.1.1 Leaky bucket

L'algorithme du *leaky bucket*, "seau percé" en français, permet de garantir un flux de sortie à un débit fixé. Nous pouvons visualiser le fonctionnement de l'algorithme en imaginant un seau d'eau percé en son fond, qui représentera le débit. Quand une goutte d'eau arrive dans le seau, s'il n'est pas plein, elle s'ajoute aux gouttes existantes puis s'écoule par le trou. Sinon, la goutte déborde et est perdue.

Ce principe peut s'appliquer à notre domaine. Si une requête arrive et que le *pool* de requêtes à traiter n'est pas plein, elle s'y rajoute et est traitée. Cependant, s'il est plein, la requête est refusée et perdue.

Cet algorithme est notamment utilisé par le serveur web NGINX [5].

4.3.1.2 Token bucket

L'algorithme du *token bucket*, "seau à jetons" en français, permet de garantir un flux de sortie flexible contrairement au *leaky bucket*. Pour faire l'analogie avec un seau, dans le cas du *token bucket* nous pouvons imaginer un seau dans lequel des jetons sont insérés à intervalle régulière. Lorsque le seau a atteint sa capacité maximale de jetons, ceux-ci ne sont simplement plus insérés.

Quand une requête arrive et qu'un jeton est disponible, celui-ci est retiré du seau et cette dernière est traitée. Par contre, s'il n'y a pas de jeton, il est possible d'imaginer deux options : ignorer la requête ou la mettre dans une file d'attente, où elle pourra être traitée lorsqu'un jeton sera de nouveau disponible. En cas d'utilisation de la seconde méthode, il faut être attentif à la taille de la file à créer. Les jetons nouvellement disponibles peuvent se retrouver complètement accaparés par les requêtes en attente et cela peut prétériter les nouvelles demandes.

4.3.1.3 Fixed window

Cet algorithme fonctionne sur un principe différent des deux précémment exposés. Il s'appuie sur une division de la temporalité en plusieurs fenêtres fixes et leur assigne un compteur. Chaque requête reçue augmente le compteur de sa fenêtre, et, lorsque celui-ci dépasse le seuil défini, les requêtes surnuméraires sont rejetées jusqu'au démarrage de la prochaine fenêtre.

Dans la figure ci-dessous, nous présentons une situation avec une *fixed window* autorisant trois requêtes par minute. À l'intérieur de la première fenêtre, entre `00:00` et `00:59`, elles arrivent progressivement jusqu'à atteindre le seuil défini. Toutes les requêtes arrivant ensuite sont refusées, comme nous pouvons le voir dans la figure 4.2. Au départ de la fenêtre suivante, le compteur est remis à zéro. Dans ce cas, de nombreuses demandes arrivent simultanément en début de fenêtre, et par conséquent les suivantes sont bloquées jusqu'au démarrage de la prochaine minute.

5. https://www.nginx.com/blog/rate-limiting-nginx/

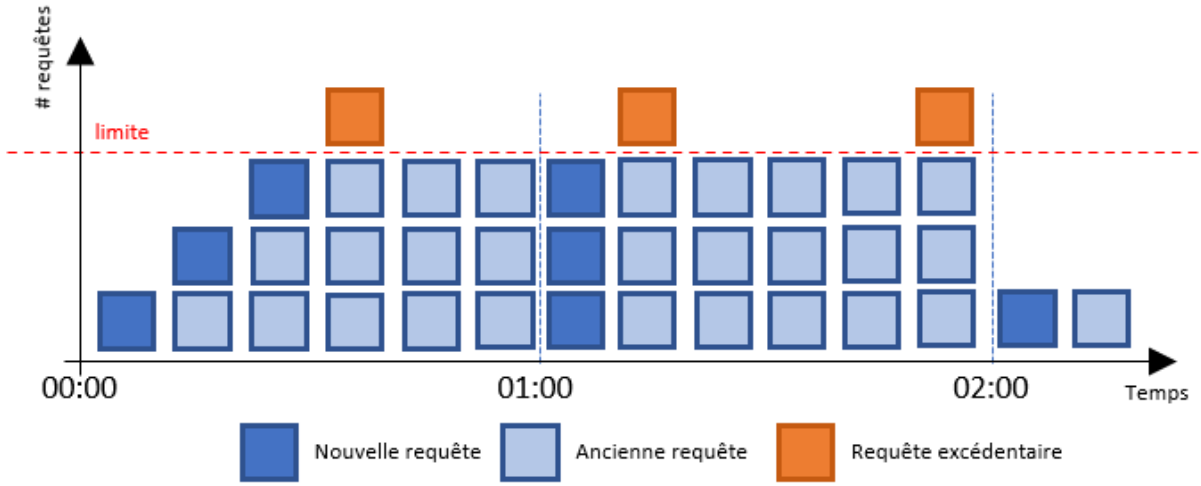

FIGURE 4.2 – *Fixed window - Exemple de traitement*
Source: *de l'auteur*

Une faiblesse de cet algorithme peut se manifester en cas de concentration des requêtes autour des limites des fenêtres, comme constaté dans l'exemple ci-dessus à `01:00`. Si un nombre important de requêtes, toutefois inférieur au seuil, arrive en fin de fenêtre, et qu'un nombre également important arrive au début de la fenêtre suivante, le serveur doit à gérer une charge importante dans un court laps de temps.

4.3.1.4 Sliding window

L'algorithme de *sliding window* cherche à résoudre un des points faible de sa variante à fenêtre fixe expliquée précédemment. La fenêtre est dynamique, prenant en considération ce qui s'est passé dans la fenêtre précédente.

Prenons par exemple une fenêtre de 60 secondes, dans laquelle nous acceptons 100 requêtes. Durant la fenêtre A, le limiteur accepte 100 requêtes. Désormais à 15 secondes dans la fenêtre suivante B, qui a reçu 30 requêtes jusqu'à présent, nous avons donc consommé $\frac{15}{60} = 0.25 = 25\%$ du temps de la fenêtre. Il faut donc prendre en compte $1 - 0.25 = 0.75 = 75\%$ de la fenêtre précédente, où 100 requêtes avaient été traitées. Pour savoir s'il est possible prendre en charge une nouvelle requête, la formule suivante est utilisée.

$$\begin{aligned} requetesTraiteesSlidingWindow &= requetesFenetreA * taux + requetesFenetreB \\ &= 100 * 0.75 + 30 \\ &= 105 \end{aligned} \tag{4.1}$$

La requête sera donc refusée, le total sur la fenêtre étant supérieur à la limite de 100.

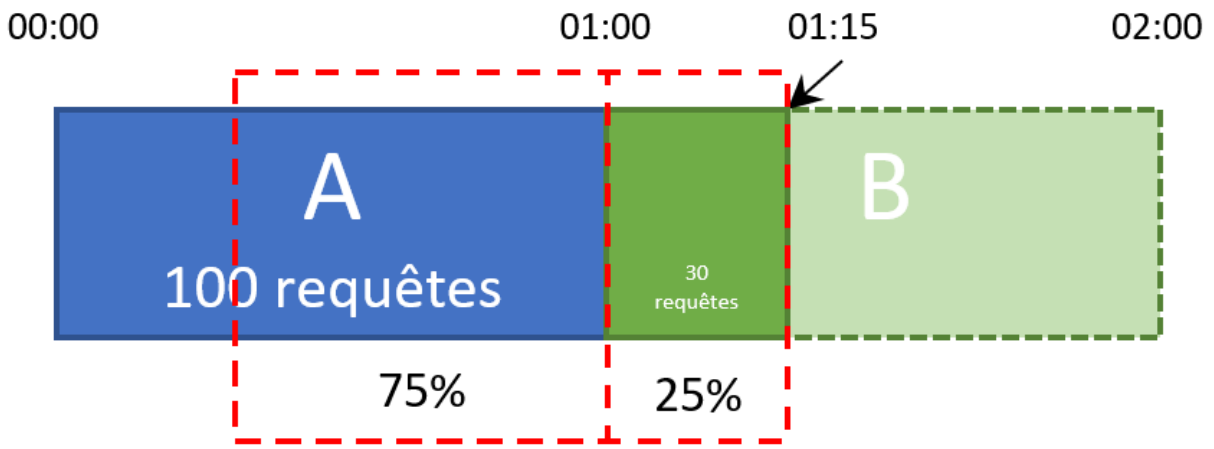

FIGURE 4.3 – *Sliding window - Requête refusée*
Source: *de l'auteur*

Quelques secondes plus tard, maintenant à 30 secondes dans notre fenêtre B, 50% du temps disponible est consommé. Notre formule ci-dessus nous donne un résultat de 80 $(100 * 0.5 + 30)$, ce qui est en dessous du seuil de 100, la requête sera donc traitée. Nous pouvons constater par cet exemple que l'algorithme de *sliding window* propose une alternative à la faiblesse présente dans l'algorithme *fixed window*.

4.3.1.5 Sélection d'un algorithme

Dans le cadre de ce travail, une solution implémentant l'algorithme *sliding window* a été retenue. Nous avons été intéressés par sa capacité à lisser les pics de requêtes ainsi que sa simplicité de mise en oeuvre. Cloudflare, une entreprise proposant un *content delivery network* ainsi que des services de protection contre les attaques par déni de service, a intégré cet algorithme dans sa solution et a partagé un retour d'expérience dans un article de leur blog [6]. Sur une analyse de 400 millions de requêtes provenant de 270'000 sources distinctes :

— 0.003% des requêtes ont été faussement autorisées, ou faussement bloquées,

— une différence de 6% entre le taux approximatif calculé par l'algorithme et le taux réel.

Nous avons donc ici un exemple d'une implémentation robuste de cet algorithme dans le cadre d'un projet à grande échelle.

4.3.2 Comparaison des algorithmes et bibliothèques *open-source*

Le besoin de gérer la charge traitée par le serveur étant une fonctionnalité importante, de nombreuses bibliothèques ont été développées. Selon les critères présentés lors du chapitre d'introduction concernant la méthodologie, les bibliothèques suivantes ont été retenues :

6. https://blog.cloudflare.com/counting-things-a-lot-of-different-things/

Bibliothèque	Doc.	Simplicité	Algorithme(s)	Bibl. externes	Dernier commit[a]
RussellLuo/slidingwindow[b]	+++	+++	*Sliding window*	0	28.05.2020
limscoder/grpc-athrottle[c]	+++	+	*Sliding window*	4	15.10.2018
mennanov/limiters[d]	++	+	*Leaky bucket* *Token bucket* *Fixed window* *Sliding window*	40	04.01.2022

[a] au 9 août 2022
[b] https://github.com/RussellLuo/slidingwindow
[c] https://github.com/limscoder/grpc-athrottle
[d] https://github.com/mennanov/limiters

TABLE 4.1 – *comparaison des bibliothèques open-source*
Source: *de l'auteur*

4.3.3 Sélection d'une solution

Selon les critères ci-dessus, nous avons retenu la bibliothèque `RussellLuo/slidingwindow`, qui comme son nom l'indique, implémente l'algorithme *sliding window*. Nous avons trouvé sa documentation très complète et précise, avec des informations concernant l'installation, ainsi qu'une explication concernant l'algorithme utilisé. Son utilisation est simple, avec la configuration d'un `Limiter` avec, en option, l'unité temporelle et la taille de la fenêtre. Nous avons également été surpris par l'absence d'utilisation de bibliothèques externes, ce qui garantit une stabilité au projet, ainsi qu'une protection contre une *supply chain attack*.

4.3.4 Implémentation

L'implémentation de la bibliothèque sélectionnée s'est effectué de la manière présentée ci-dessous. Le *middleware* se retrouve dans le fichier `internal\middleware\throttling.go`

```
1       func AddRateLimiterServerInterceptor(limiter *sw.Limiter,
    ↪ treshold int, logger *zap.Logger) grpc.UnaryServerInterceptor {
2               lim, _ := sw.NewLimiter(time.Minute, int64(treshold),
    ↪ func() (sw.Window, sw.StopFunc) {
3               return sw.NewLocalWindow()
4       })
5       return func(ctx context.Context, req interface{}, info
    ↪ *grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{},
    ↪ error) {
6               if ok := lim.Allow(); !ok {
7                       logger.Warn("request rejected by rate limiter",
    ↪ zap.String("info", info.FullMethod))
8                       return nil,
    ↪ status.Errorf(codes.ResourceExhausted, "request %s rejected by rate
    ↪ limiter", info.FullMethod)
```

```
                }
                return handler(ctx, req)
        }
}
```

L'*interceptor* peut ensuite être intégré au serveur :

```
if throttling {
                logger.Log.Info("Enabling throttling")
                interceptor = append(interceptor,
                        middleware.AddRateLimiterServerInterceptor(&slidi
↪  ngwindow.Limiter{},
↪  logger.Log),
                )
        }
```

Pour tester le bon fonctionnement du limiteur, une application cliente adaptée pour envoyer un grand nombre de requêtes durant une courte période a été dévelopée.

```
client := pb.NewHelloWorldClient(conn)
for i := 0; i < 1000; i++ {
        var header metadata.MD

        // On appelle le service SayHello avec une HelloRequest
        reply, err := client.SayHello(context.Background(),
↪  &pb.HelloRequest{Name: "David"}, grpc.Header(&header))
        // En cas d'erreur
        if err != nil {
                log.Print(err.Error())
        } else {
                // On affiche la réponse du serveur
                log.Printf("Hello response from server %s // %s",
↪  reply.Greetings, header.Get("correlation-id")[0])
        }
}
```

L'exécution du code ci-dessus permet d'obtenir la trace suivante dans la console du serveur :

```
Connecting to gRPC server
Hello response from server Hello David!
Hello response from server Hello David!
Hello response from server Hello David!
Hello response from server Hello David!
Hello response from server Hello David!
rpc error: code = ResourceExhausted desc = request
↪  /v1.HelloWorld/SayHello rejected by rate limiter
```

```
rpc error: code = ResourceExhausted desc = request
↪  /v1.HelloWorld/SayHello rejected by rate limiter
rpc error: code = ResourceExhausted desc = request
↪  /v1.HelloWorld/SayHello rejected by rate limiter
```

Le limiteur a été configuré pour que sa fenêtre ait une taille de 5, et nous pouvons constater qu'effectivement 5 requêtes ont été traitées, et les suivantes ont été rejetées.

4.4 Chiffrement des données

4.4.1 État de l'art

Le cahier des charges de ce travail demandait de pouvoir assurer une communication sécurisée entre le client et le serveur, au travers du chiffrement à l'aide du protocole TLS. Différents modes ont été demandés : pas de TLS, TLS côté serveur et TLS mutuel.

Nous allons présenter ici quelques aspects nécessaires à la compréhension de ce qui a été implémenté.

4.4.1.1 SSL/TLS

SSL, pour *Secured Sockets Layer*, est un protocole qui a pour but d'établir des connexions authentifiées et sécurisées entre des éléments d'un réseau informatique. Son successeur, TLS, pour *Transport Layer Security*, a été publiée en 1999 dans sa version 1.0. Actuellement, le protocole se trouve dans sa version 1.3 depuis 2018, exprimée dans la RFC 8446 [7].

4.4.1.2 No TLS

Lors d'une communication sans TLS, les échanges entre le client et le serveur sont effectués en clair. Ils ne sont pas sécurisés, en cas d'interception par un tiers son contenu pourra être lu sans efforts particuliers.

Les figures ci-dessous représentent la trame d'échange entre un client et un serveur, analysée à l'aide du programme Wireshark.

7. https://datatracker.ietf.org/doc/html/rfc8446

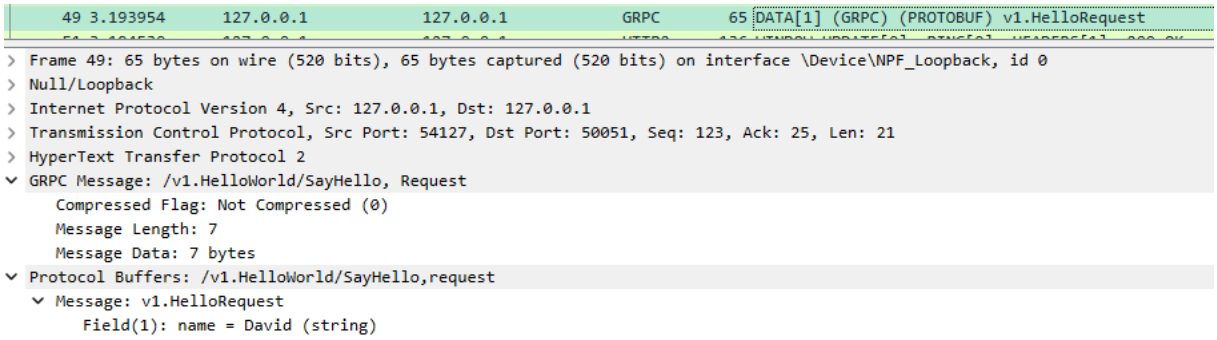

FIGURE 4.4 – *Requête du client, avec le paramètre en clair*
Source: *de l'auteur*

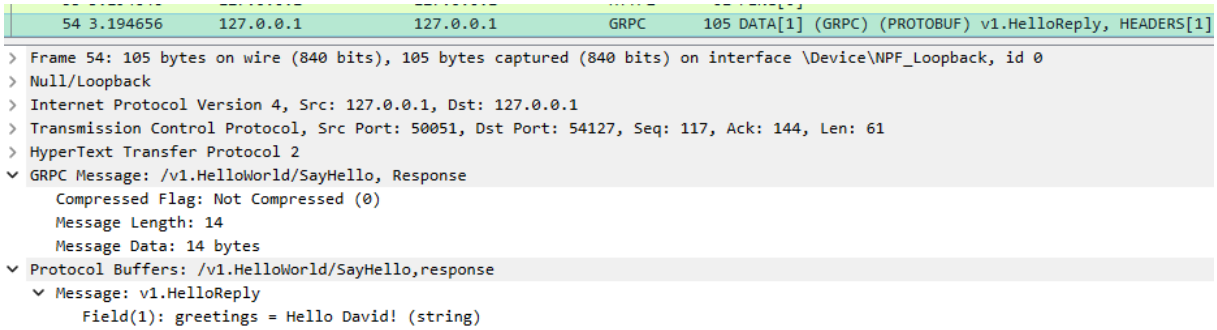

FIGURE 4.5 – *Réponse du serveur en clair*
Source: *de l'auteur*

4.4.1.3 Server-side TLS

Dans ce cas de figure, seul le serveur doit prouver son identité. Il doit fournir son certificat au client, afin que celui-ci puisse valider son authenticité. Les étapes décrites ci-dessous correspondent à la version 1.3 du protocole.

Le processus, appelé TLS *handshake*, est le suivant :

1. Le client envoie au serveur la version maximale de TLS supportée, les algorithmes de chiffrement pris en charge (*cipher suite*), ainsi qu'un nombre aléatoire. Cet échange est appelé *ClientHello*.
2. Le serveur répond avec le *ServerHello*, en renvoyant la version du protocole qui sera utilisée, la *cipher suite* sélectionnée, ainsi qu'un nombre aléatoire.
3. Le serveur envoie son certificat, puis un *ServerHelloDone* pour indiquer que sa partie du *handshake* est terminée.
4. Le client répond avec un message *ClientKeyExchange*, où il va envoyer un *PreMasterSecret* encrypté à l'aide de la clé publique du certificat du serveur.
5. Le client et le serveur vont désormais utiliser les nombres aléatoires échangés ainsi que le *PreMasterSecret* pour générer un secret commun appelé *master secret* qui servira de base aux communications futures.

6. Le client envoie un *ChangeCipherSpec* pour indiquer que les communications sont à présent authentifiées. Il envoie un message *Finished* sécurisé au serveur.
7. Le serveur reçoit le message *Finished* et tente de le valider sur la base des informations échangées précédemment. Si tout s'est déroulé correctement, il envoie également son *ChangeCipherSpec* contenant un message *Finished* au client.
8. Le client valide de la même manière le message *Finished* du serveur.
9. Le *handshake* est terminé, les échanges applicatifs sont désormais authentifiés et sécurisés.

Pour comparer avec l'absence de TLS du cas précédent, nous pouvons effectuer une nouvelle analyse avec Wireshark et constater que les échanges ne sont plus en clair.

127.0.0.1	127.0.0.1	TLSv1.3	314		Client Hello
127.0.0.1	127.0.0.1	TLSv1.3	2412	692358c3915755bdcd7cb07…	Server Hello, Change Ciph
127.0.0.1	127.0.0.1	TLSv1.3	108	dd396f5e12dd10007fc94f5…	Change Cipher Spec, Appli
127.0.0.1	127.0.0.1	TLSv1.3	90	f217f749f607cabc55ad72f…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	75	793ea788a57f662a317cb3b…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	81	276a24c05db62723148765e…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	75	9877ae48ee731556dfdbbe3…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	75	091165ac07722de00cd2c38…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	146	c81da002ae7fca1b815b9e2…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	87	ec9bdf0531403b4f48816b1…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	96	63b13cdef9e31581a14e33c…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	83	3eeef343a472918f99fc6a8…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	189	d3b208e1918c135e050eb61…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	96	032fe085a75927f0abdefb0…	Application Data
127.0.0.1	127.0.0.1	TLSv1.3	83	6312c713603fcb735aecf5c…	Application Data

FIGURE 4.6 – *Echanges entre le client et le serveur*
Source: *de l'auteur*

4.4.1.4 Mutual TLS

Dans le cas de figure du TLS mutuel, ou *two-way TLS*, le client doit également fournir un certificat. Ces étapes concernent également la version 1.3 du protocole.

Le Secure Sockets Layer (SSL) *handshake* est similaire au précédent, avec toutefois quelques étapes supplémentaires :

1. Le client envoie au serveur la version maximale de SSL/TLS supportée ainsi que les algorithmes de chiffrement pris en charge (*cipher suite*), ainsi qu'un nombre aléatoire. Cet échange est appelé *ClientHello*.
2. Le serveur répond avec le *ServerHello*, en renvoyant la version du protocole qui sera utilisée, la *cipher suite* sélectionnée, ainsi qu'un nombre aléatoire.
3. Le serveur envoie son certificat.
4. Le serveur demande au client son certificat avec le message *CertificateRequest*, puis un *ServerHelloDone* pour indiquer que sa partie du *handshake* est terminée.
5. Le client envoie son certificat au serveur via le message *Certificate*.

6. Le client répond avec un message *ClientKeyExchange*, où il va envoyer un *PreMasterSecret* encrypté à l'aide de la clé publique du certificat du serveur.
7. Il envoie un *CertificateVerify* qui correspond à un chiffrement des échanges précédents à l'aide de la clé privée du client. Le serveur ayant accès à la clé publique au travers du certificat du client, il peut confirmer que le certificat est bien le sien.
8. Le client et le serveur utilisent les nombres aléatoires échangés ainsi que le *PreMasterSecret* pour générer un secret commun appelé *master secret* qui servira de base aux communications futures.
9. Le client envoie un *ChangeCipherSpec* pour indiquer que les communications sont dès à présent authentifiées. Il envoie un message *Finished* sécurisé au serveur.
10. Le serveur reçoit le message *Finished* et tente de le valider sur la base des informations échangées précédemment. Si tout s'est déroulé correctement, il envoie également son *ChangeCipherSpec* contenant un message *Finished* au client.
11. Le client valide de la même manière le message *Finished* du serveur.
12. Le *handshake* est terminé, les échanges applicatifs sont désormais authentifiés et sécurisés.

4.4.2 Comparaison des algorithmes et bibliothèques *open-source* et sélection d'une solution

La prise en charge des échanges TLS étant intégrée dans la bibliothèque standard `crypto/tls`, il n'a pas été nécessaire d'en chercher une externe proposée par la communauté. La version 1.3 de TLS est prise en charge depuis la version 1.12 de Go en mode *opt-in*[8], puis par défaut dès la version 1.13[9].

4.4.3 Implémentation

Pour faciliter le processus, un script a été développé pour permettre la génération des différents certificats nécessaires : authorité de certification (CA), serveur et client. Il est disponible dans les annexes de ce document.

Les différents éléments utiles à la gestion du TLS ont été définis dans un fichier `tls.go`. Afin de prendre en charge différents modes de communication, un `type` a été défini, ainsi que des constantes correspondantes. Elles pourront être ensuite appelées dans le code côté client ou serveur afin de déterminer le comportement à adopter.

```
type Tls_mode string

const (
        TLS_None    Tls_mode = "TLS_NONE"
```

8. https://pkg.go.dev/crypto/tls@go1.12
9. https://pkg.go.dev/crypto/tls@go1.13

```
        TLS_Server Tls_mode = "TLS_SERVER"
        TLS_Mutual Tls_mode = "TLS_MUTUAL"
)
```

Une structure de paramétrage a également été définie pour faciliter la configuration du serveur.

```
type TlsSettings struct {
        Tls_mode    Tls_mode
        Server_cert string
        Server_key  string
        Ca_cert     string
}
```

Lors du démarrage du serveur, le comportement à adopter est défini en fonction du mode voulu par l'utilisateur dans la structure `TlsSettings`.

```
1  // Gestion des paramètres TLS (grpc + grpc gateway)
2  if tls_settings.Tls_mode == TLS_Server || tls_settings.Tls_mode ==
   ↪  TLS_Mutual {
3          if tls_settings.Tls_mode == TLS_Server {
4                  tlsCredentials, tls_err =
   ↪  loadTLSCredentials(tls_settings.Tls_mode, tls_settings.Server_cert,
   ↪  tls_settings.Server_key, "")
5
6          } else {
7                  tlsCredentials, tls_err =
   ↪  loadTLSCredentials(tls_settings.Tls_mode, tls_settings.Server_cert,
   ↪  tls_settings.Server_key, tls_settings.Ca_cert)
8          }
9          if tls_err != nil {
10                 logger.Log.Error("cannot load TLS credentials: ")
11                 logger.Log.Error(err.Error())
12         }
13         serverOptions = append(serverOptions, grpc.Creds(tlsCredentials))
14
15         // Gestion TLS pour la gateway également
16         loadTLSCredentialsGateway, tls_err :=
   ↪  loadTLSCredentialsGateway(tls_settings.Tls_mode)
17         if tls_err != nil {
18                 logger.Log.Error("cannot load TLS credentials for
   ↪  gateway: ")
19                 logger.Log.Error(err.Error())
20         }
21         dialOptions = append(dialOptions,
   ↪  grpc.WithTransportCredentials(loadTLSCredentialsGateway))
22 } else {
```

```
        // En cas de NoTLS, on autorise les connexions non sécurisées
        serverOptions = append(serverOptions,
    ↪ grpc.Creds(insecure.NewCredentials()))
        dialOptions = append(dialOptions,
    ↪ grpc.WithTransportCredentials(insecure.NewCredentials()))
}
```

Finalement, lors du démarrage de la *gateway*, il est nécessaire d'adapter le comportement en cas de connexions sécurisées :

```
if tls_settings.Tls_mode == TLS_Server || tls_settings.Tls_mode ==
    ↪ TLS_Mutual {
        zap.S().Info("Serving gRPC-Gateway on https://" + serverIp + ":"
    ↪ + grpcGwPort)
        log.Fatalln(gwServer.ListenAndServeTLS(crt, key))
} else {
        zap.S().Info("Serving gRPC-Gateway on http://" + serverIp + ":" +
    ↪ grpcGwPort)
        log.Fatalln(gwServer.ListenAndServe())
}
```

Après avoir installé le certificat de l'autorité de certification dans le navigateur web, nous pouvons constater que la connexion est sécurisée, et le certificat valide.

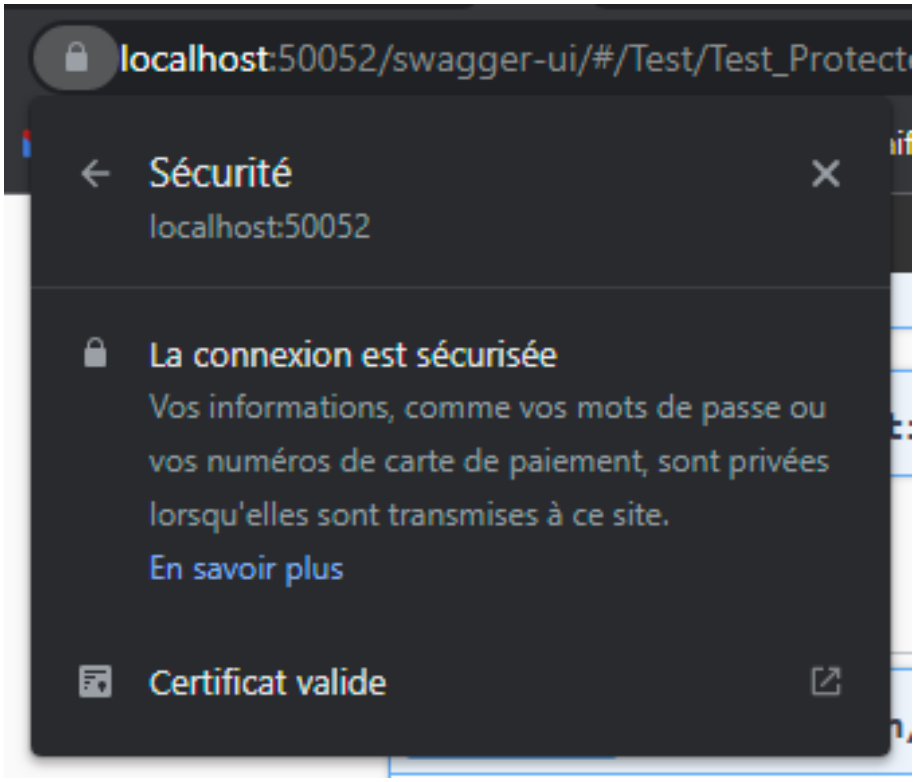

FIGURE 4.7 – *Communication sécurisée avec Swagger UI*
Source: *de l'auteur*

4.5 Authentification

4.5.1 État de l'art

Selon le cahier des charges, il est demandé de pouvoir gérer l'authentification à l'aide de *tokens* JWT de différents *identity providers*. Après discussion avec le mandant, le travail s'est concentré sur ceux générés par Azure et Google. L'état de l'art concernant les *tokens* JWT sera décrit dans le chapitre suivant concernant les sessions (section 4.6).

4.5.1.1 Microsoft Azure

Dans ce projet, il est possible de s'authentifer via un *Active Directory Azure*. C'est une fonctionnalité *cloud* offerte par Microsoft qui fournit des services de gestion d'identité et d'accès. Un projet a été crée dans l'*Active Directory Azure* HES-SO [10] qui est mis à disposition des étudiant·e·s. La connexion se fait donc à l'aide d'un compte `@hes-so.ch`. Le détail de la mise en place du projet se trouve dans les annexes à la fin de ce document.

Pour obtenir un *token*, le logiciel Postman [11] qui permet d'interagir avec des API a été utilisé.

Nous pouvons effectuer une requête `POST` sur une URL propre à ce projet, avec différents paramètres passés via le *body*. Par la suite, nous pourrons ajouter le *token* obtenu aux *headers* d'une requête à notre serveur afin de valider le *token* et accéder à différents services.

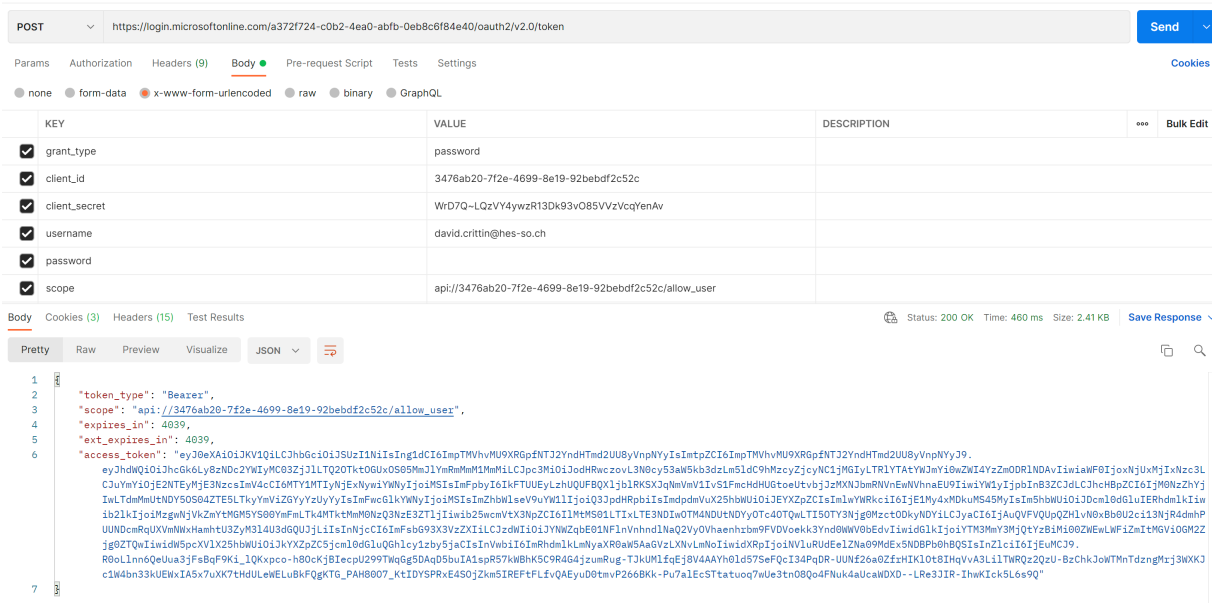

FIGURE 4.8 – *Requête de token Azure Active Directory avec Postman*
Source: *de l'auteur*

10. https://portal.azure.com/
11. https://www.postman.com/

4.5.1.2 Google Cloud Console

Afin de pouvoir s'authentifier à l'aide d'un compte Google, nous avons créé un projet dans la console Google Cloud Platform [12]. Comme pour le projet Azure, la marche à suivre est disponible dans les annexes de ce document.

Pour obtenir le *token* qui sera utilisé dans l'application, nous avons également du configurer une requête Postman afin d'obtenir un *token* auprès de Google.

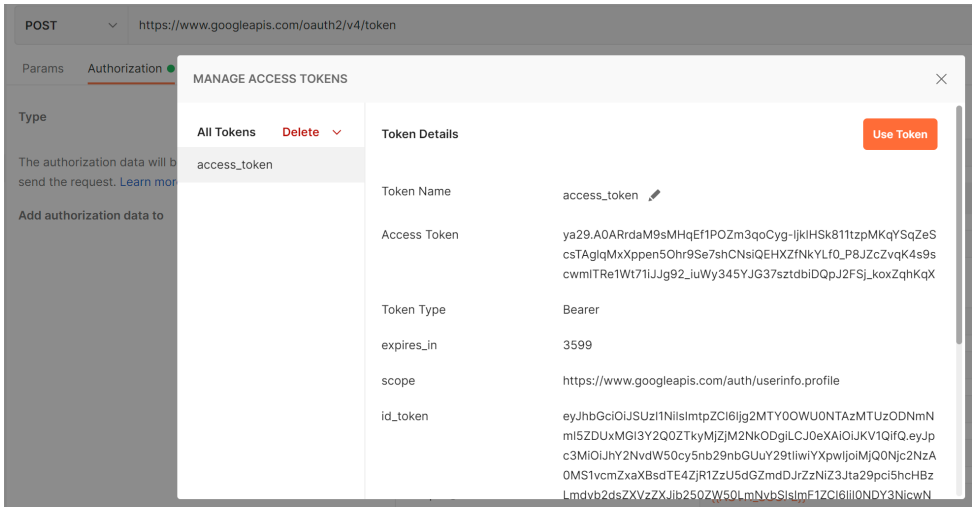

FIGURE 4.9 – *Requête de token Google avec Postman*
Source: *de l'auteur*

4.5.2 Comparaison des algorithmes et bibliothèques *open-source*

Nous avons besoin ici d'une bibliothèque permettant de travailler avec des *tokens* JWT. Comme lors des précédents chapitres, un panel de critères est utilisé pour sélectionner une bibliothèque a intégrer dans le projet. Celui concernant l'algorithme proposé n'est pas retenu ici, comme il n'est pas pertinent dans ce cas.

Bibliothèque	Doc.	Simplicité	Bibl. externes	Dernier commit[a]
golang-jwt/jwt[b]	+++	+++	0	04.06.2022
robbert229/jwt[c]	+	+	*pas de go.mod*	08.11.2018

[a] au 9 août 2022
[b] https://github.com/golang-jwt/jwt
[c] https://github.com/robbert229/jwt

TABLE 4.2 – *Comparaison des bibliothèques open-source*
Source: *de l'auteur*

12. https://console.cloud.google.com/

4.5.3 Sélection d'une solution

Selon ce que nous avons constaté suite à notre recherche de bibliothèques et au différents fragment de code parcourus, une grande majorité utilise la bibliothèque *golang-jwt/jwt*. Nous avons pu trouver également quelques références au package *dgrijalva/jwt-go*[13] qui a été déprécié en juin 2021 au profit du *repository* précédent.

Au vu de la grande adhésion de la communauté à cette bibliothèque, nous avons décidé de l'utiliser dans notre projet. Elle est régulièrement mise à jour, et la documentation est de grande qualité, avec de nombreux exemples mis à disposition.

4.5.4 Implémentation

Nous devons pouvoir valider les *tokens* des deux *providers* sélectionnés, à savoir Azure et Google. Afin de tester l'authentification, deux routes ont été mises en place, `/v1/authenticate/azure` et `/v1/authenticate/google`, qui permettent de vérifier les *tokens*.

Ces deux routes appellent leur méthode d'authentification respective, à savoir `AzureAuth` et `GoogleAuth`, qui appellent `Authenticate` avec des paramètres propres.

```
1  // Authentification avec JWT Azure
2  func (s *server) AzureAuth(ctx context.Context, in
   ↪  *pb.AuthenticateRequest) (*pb.AuthenticateResponse, error) {
3      return s.Authenticate(pb.Provider_PROVIDER_AZURE, ctx, in)
4  }
5
6  // Authentification avec JWT Google
7  func (s *server) GoogleAuth(ctx context.Context, in
   ↪  *pb.AuthenticateRequest) (*pb.AuthenticateResponse, error) {
8      return s.Authenticate(pb.Provider_PROVIDER_GOOGLE, ctx, in)
9  }
```

Afin de valider les *tokens* des *providers*, il est nécessaire de récupérer les clés publiques de ces derniers. Elles sont appelées *JSON Web Key Set*, ou JWKS. Les adresses de ces JWKS sont stockées dans des variables :

```
1  var (
2      azureJwksUrl  =
   ↪  "https://login.microsoftonline.com/common/discovery/keys"
3      googleJwksUrl = "https://www.googleapis.com/oauth2/v3/certs"
4  )
```

Ensuite, lors de l'appel à la méthode principale *Authenticate*, le contenu du *key set* est téléchargé afin de valider la signature du *token*.

13. https://github.com/dgrijalva/jwt-go

```
1  func (s *server) Authenticate(provider pb.Provider, ctx context.Context,
   ↪  in *pb.AuthenticateRequest) (*pb.AuthenticateResponse, error) {
2  ...
3      var targetKeyFunc jwt.Keyfunc
4      if provider == pb.Provider_PROVIDER_AZURE {
5          targetKeyFunc = getKeyAzure
6      } else if provider == pb.Provider_PROVIDER_GOOGLE {
7          targetKeyFunc = getKeyGoogle
8      } else {
9          return &pb.AuthenticateResponse{IsValid: false, Message: "Unknown
   ↪  provider"}, nil
10     }
11     token, err := jwt.Parse(jwtFromHeader, targetKeyFunc)
12     if err != nil {
13         return &pb.AuthenticateResponse{IsValid: false, Message:
   ↪  err.Error()}, nil
14     } else {
15         if token.Valid {
16             return &pb.AuthenticateResponse{IsValid: true}, nil
17
18         } else {
19             logger.Log.Sugar().Infof("Error : %s", err.Error())
20         }
21     }
22 ...
23 }
```

Nous pouvons ensuite appeler la route voulue, par exemple celle concernant Azure, avec le *token* obtenu via Postman. Pour le navigateur Google Chrome, l'extension *ModHeader*[14] rajoute un *request header* de type `Authorization`.

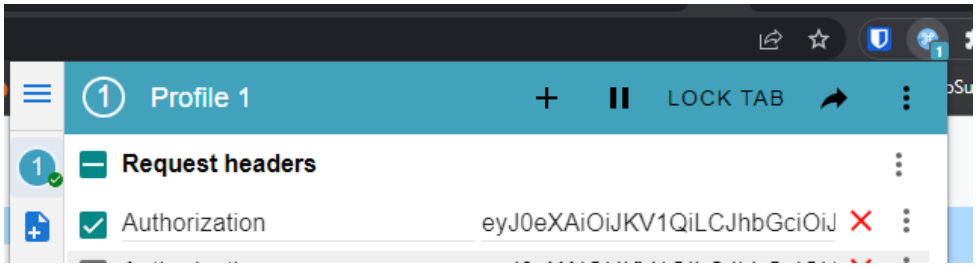

FIGURE 4.10 – *Ajout de token via ModHeader*
Source: *de l'auteur*

Dans Swagger UI, il est possible d'appeler la route afin de contrôler la validité du *token*.

14. https://modheader.com/install

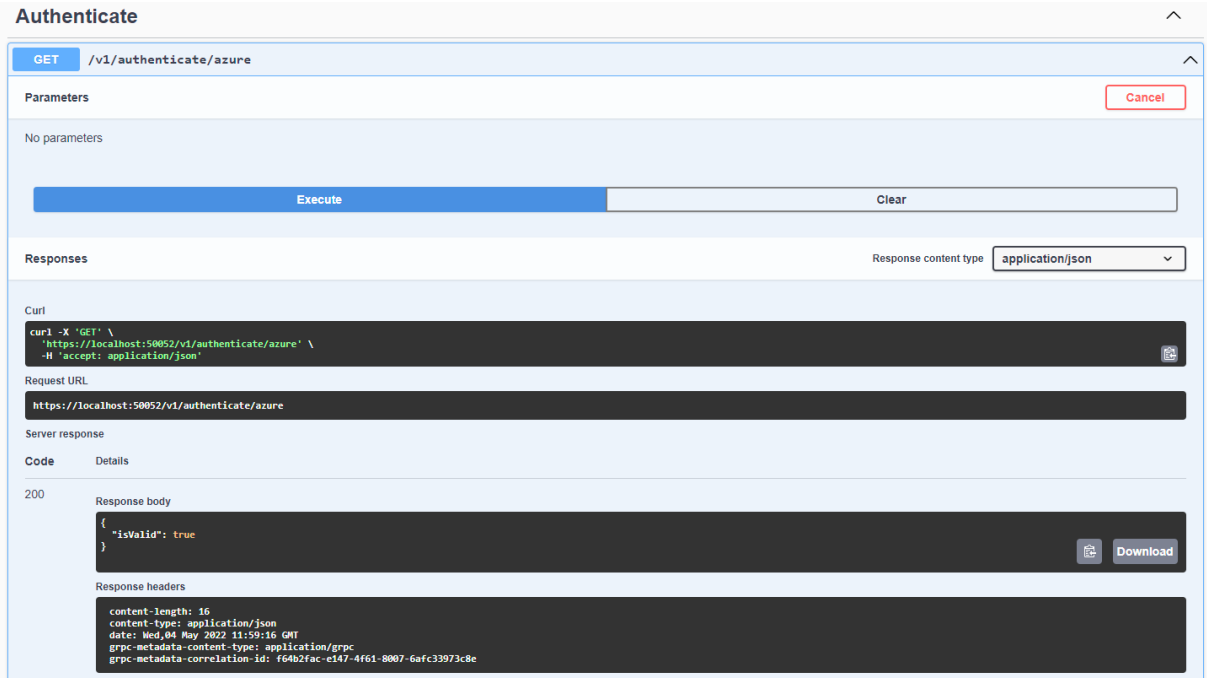

FIGURE 4.11 – *Appel de la route de validation Azure avec Swagger UI*
Source: *de l'auteur*

Il est également possible d'effectuer l'appel via Postman, selon la capture ci-dessous.

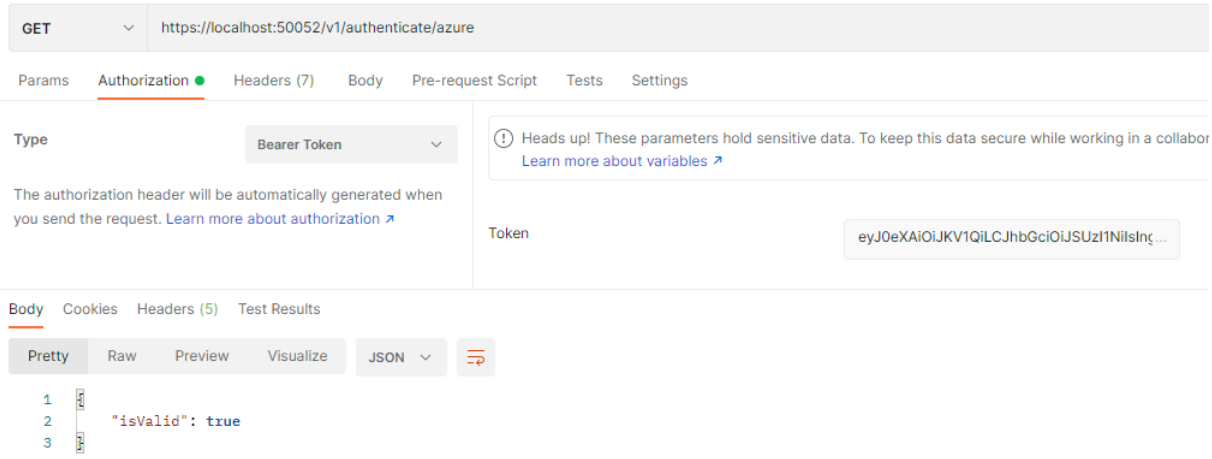

FIGURE 4.12 – *Appel de la route de validation Azure avec Postman*
Source: *de l'auteur*

4.6 Gestion des sessions

4.6.1 État de l'art

Une fois l'authentification effectuée auprès d'un des services externes présentés au chapitre précédent, le serveur doit pouvoir fournir des *tokens* JWT.

JSON Web Token

Le JSON *Web Token*, abrégé JWT dans notre document, est un moyen standardisé au travers de la RFC 7519 [15] d'échanger des informations de manières sécurisée en tant qu'objet JSON. Un *token* JWT est composé de trois parties : *header*, *payload* et *signature*. Chaque partie du *token* représente des informations précises. Le *token* est une suite de caractères, séparées par des «.».

— *header* contient les informations relatives à l'algorithme utilisé pour générer la signature. Il stocke également l'information sur le type de *token*, dans ce projet cas JWT. Différents algorithmes sont disponibles selon la RFC.

— *payload* va contenir les *claims*, à savoir des pièces d'informations sécurisées à propos d'un sujet. Elles fonctionnent en paire clé/valeur. La RFC définit dans sa section 10.1 plusieurs *claims* standards qui pourront être utilisés. Il est également possible de définir des *custom claims* pour répondre à un besoin spécifique.

— *signature* permet de vérifier que le message n'a pas été modifié durant la vie du *token*. Elle est calculée en encodant le *header* et le *payload* en *base64* et en concaténant le resultant séparé par un «.». Le tout est ensuite signé par l'algorithme indiqué dans le *header*.

L'ensemble de ces trois parties va donner un chaîne de caractères encodée en *base64*. Prenons par exemple le *token* JWT suivant :

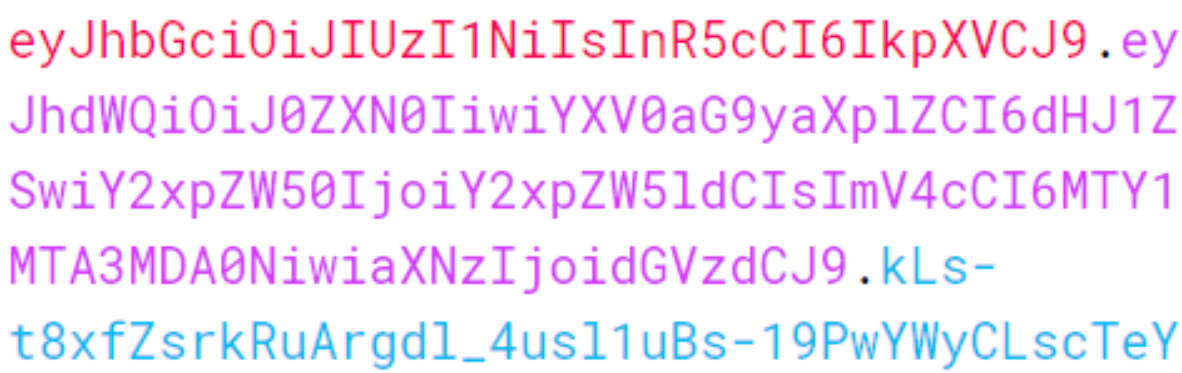

FIGURE 4.13 – *Exemple de token JWT*
Source: *de l'auteur*

La partie en rouge, représentant le *header*, contient les informations suivantes :

```
{
        "alg": "HS256",
```

15. https://tools.ietf.org/html/rfc7519

```
        "typ": "JWT"
}
```

On retrouve l'algorithme utilisé, ici HMAC + SHA256 et le type de *token*.

Le contenu décodé de la partie violette qui représente le *payload* est le suivant :

```
{
        "aud": "test",
        "authorized": true,
        "client": "client",
        "exp": 1651070046,
        "iss": "test"
}
```

Nous retrouvons ici différents *claims* standardisés : *aud* qui représente l'audience à laquelle est destinée le *token*, *exp* correspondant au *timestamp* à partir duquel le *token* n'est plus valide et *iss* qui identifie qui a émis le *token*. Il y a également deux *claims* personnalisés : *authorized* et *client* rajoutés lors de la création du *token*.

Finalement, la partie en bleu permet de valider la signature du *token*, selon la procédure décrite plus haut.

4.6.2 Comparaison des algorithmes et bibliothèques *open-source* et sélection d'une solution

Le besoin de gestion des *tokens* JWT ayant été rempli lors du précédent chapitre concernant l'authentification (sous-section 4.5.2), la même bibliothèque sera utilisée, à savoir *golang-jwt/jwt*.

4.6.3 Implémentation

Dans ce projet, nous avons décidé de signer les *tokens* JWT à l'aide d'un secret qui sera stocké dans une variable d'environnement. Dans un premier temps il doit être défini au niveau de la machine locale Unix à l'aide de la commande suivante :

```
export SECRET_KEY="supersecretkey"
```

Cette clé peut ensuite être retrouvée à l'aide de ce code :

```
var signingKey = []byte(os.Getenv("SECRET_KEY"))
```

Deux méthodes sont ensuite nécessaires, une permettant de générer un *token* JWT, et une autre permettant d'en contrôler la validité.

La première méthode se présente comme suit :

```go
func GetJWT() (string, error) {
        // Création d'un nouveau token avec la méthode HMAC + SHA-256
        token := jwt.New(jwt.SigningMethodHS256)

        // Créations des claims du token
        claims := token.Claims.(jwt.MapClaims)

        // Paramétrage de quelques claims
        // claims standardisés
        claims["aud"] = "test"
        claims["iss"] = "test"
        claims["exp"] = time.Now().Add(time.Minute * tokenDuration).Unix()
        // claims custom
        claims["authorized"] = true
        claims["client"] = "client"

        // Signature du token
        tokenString, err := token.SignedString(signingKey)

        if err != nil {
                fmt.Errorf("Something Went Wrong: %s", err.Error())
                return "", err
        }

        return tokenString, nil
}
```

La méthode permettant de contrôler la validité d'un *token* est la suivante :

```go
func validateToken(ctx context.Context) (bool, string) {
        // Récupération des metadata
        md, ok := metadata.FromIncomingContext(ctx)
        if ok {
                jwtFromHeader := ""
                // Si une metadata "authorization" est présente, on
↪  récupère la valeur
                if len(md["authorization"]) > 0 {
                        jwtFromHeader = md["authorization"][0]
                } else {
                        return false, "Could not find token"
                }
                // Récupération du token
                token, err := jwt.Parse(jwtFromHeader, func(token
↪  *jwt.Token) (interface{}, error) {
                        if _, ok :=
↪  token.Method.(*jwt.SigningMethodHMAC); !ok {
```

```
                                return nil, fmt.Errorf(("Invalid signing
↪  method"))
                        }
                        return signingKey, nil
                })
                if err != nil {
                        return false, err.Error()
                }

                // Contrôle de la validité du token
                if token.Valid {
                        return true, ""
                }
        }
        return false, ""
}
```

Nous pouvons tester le fonctionnement à l'aide de Swagger UI, ou Postman. Ci-dessous, le *token* obtenu à l'aide de Postman :

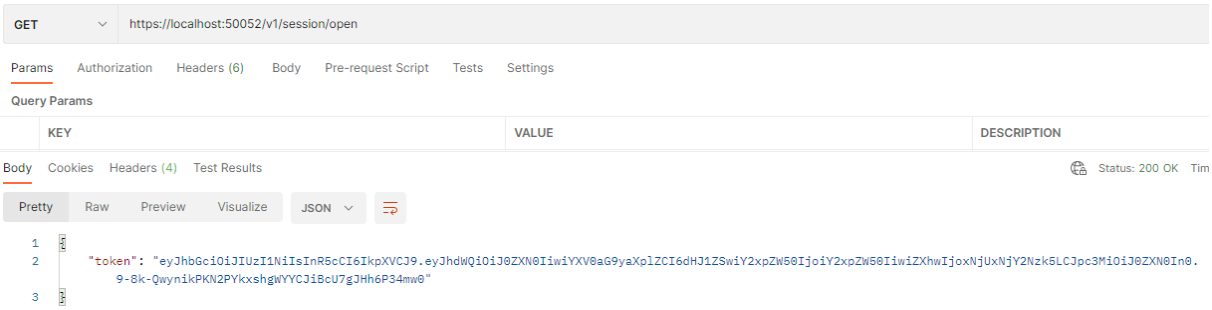

FIGURE 4.14 – *Obtention d'un token JWT avec Postman*
Source: *de l'auteur*

Il est ensuite possible de décoder le *token* obtenu à l'aide du site jwt.io [16], et vérifier que les informations du *token* correspondent à celles programmées.

16. https://jwt.io

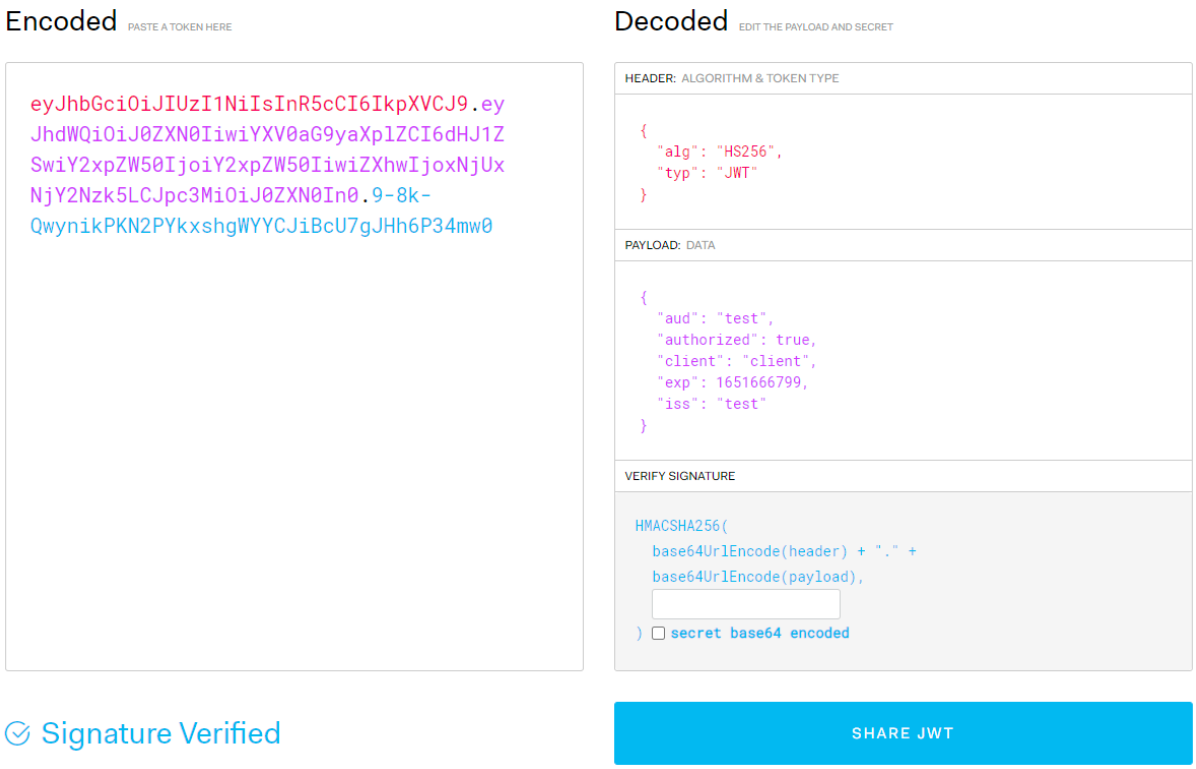

FIGURE 4.15 – *Décodage du token JWT obtenu*
Source: *de l'auteur*

5 Implémentation d'une API de chiffrement externe

5.1 Introduction

Une fois le serveur fonctionnel avec les *middlewares* requis, il nous a été demandé d'implémenter une API de chiffrement externe. Le mandant proposait l'intégration d'une des solutions actuelles du marché : Microsoft Double Key Encryption [1] ou Google Client Side Encryption [2].

Nous avons été mis en contact avec la société DuoKey [3] qui propose du *Key Management as a Service (KMaaS)* à destination du cloud. DuoKey était intéressé par la mise à disposition d'une solution basée sur Google Client Side Encryption, et nous a proposé son infrastructure Google Workspace. Les développements se sont donc concentrés sur cette solution.

5.2 Google Client Side Encryption

5.2.1 Présentation de la solution

La solution proposée par Google permet à l'utilisateur de disposer de ses propres clés de chiffrement des documents au sein d'un Google Workspace [4]. Celles-ci sont stockées dans l'infrastructure de l'utilisateur, et non chez Google. Cela permet de rendre les données indéchiffrables par le fournisseur de services *cloud*, le client ayant la main sur les clés utilisées, et ainsi de garantir la sécurité de ses documents. Actuellement, le service est en *restricted beta*, l'accès devant être demandé auprès de Google [5].

Au moment de la rédaction de ce document, il est possible de créer un document, une feuille de calcul ou une présentation chiffrée. L'utilisateur a également la possibilité d'importer et de chiffrer un document ou dossier. L'option sera disponible plus tard également pour le service de réunions en ligne Google Meet, ainsi que pour certaines autres fonctionnalités disponibles dans Google Workspace, comme par exemple Google Calendar.

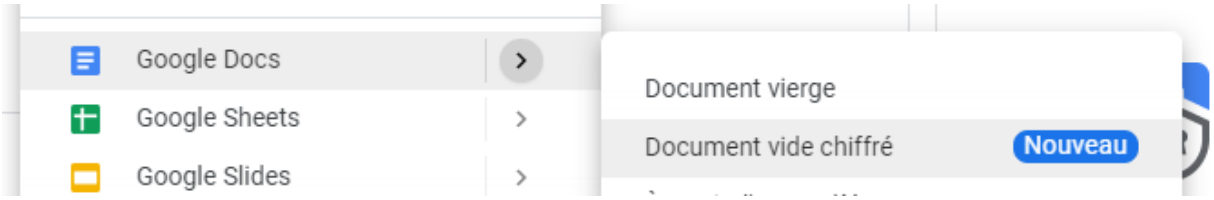

FIGURE 5.1 – *Création d'un nouveau document chiffré*
Source: *de l'auteur*

1. https://docs.microsoft.com/en-us/microsoft-365/compliance/double-key-encryption?view=o365-worldwide
2. https://developers.google.com/workspace/cse
3. https://duokey.com/
4. https://cloud.google.com/blog/products/workspace/new-google-workspace-security-features
5. au 9 août 2022

L'architecture des différents services est la suivante :

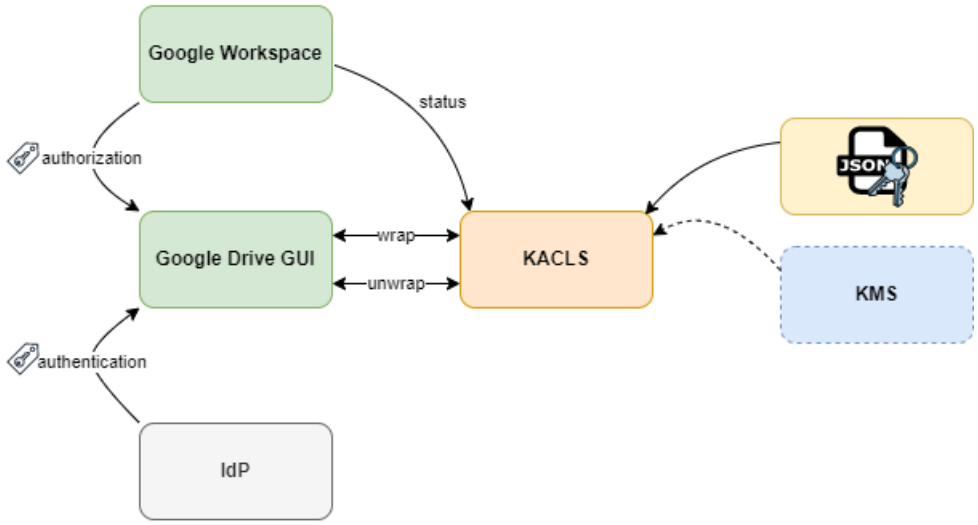

FIGURE 5.2 – *Architecture des différents services*
Source: *de l'auteur*

Nous retrouvons en point central le serveur qui remplit le rôle de *Key Access Control List Service* (KACLS). Ce dernier est hébergé chez l'utilisateur. Il s'occupe de traiter les différentes demandes, comme celles de *wrap* et *unwrap* de clés. Dans le cadre de ce projet, le *keyset* nécessaire pour la gestion du chiffrement est stockée dans un fichier JSON. Dans un cas de production, l'utilisation d'un *Key Management Service* (KMS) lui sera préférée.

5.2.2 Fonctionnement

Le processus de création d'un document chiffré est le suivant :

— L'utilisateur accède au Google Workspace de son entreprise et demande la création d'un document chiffré (figure 5.1).

— Le service demande l'authentification de l'utilisateur auprès d'un *Identity Provider* (IdP) externe.

— Une fois l'utilisateur authentifié, Google Workspace interroge le service, qui joue le rôle de KACLS, en lui transmettant une *Data Encryption Key* (DEK), un *token* JWT `authentication`, un *token* JWT `authorization` ainsi qu'une chaîne de caractères JSON `reason`. Le *token* `authentication` est fourni par l'IdP et atteste de l'identité de l'utilisateur [6], et le *token* `authorization` fourni par Google permet de savoir si l'utilisateur a le droit de chiffrer ou déchiffrer un ressource [7].

— Le KACLS effectue des vérifications sur les *tokens*, puis, si tout est valide, utilise un algorithme de chiffrement authentifié avec données additionnelles (AEAD) pour chiffrer la DEK.

6. https://developers.google.com/workspace/cse/reference/authentication-tokens
7. https://developers.google.com/workspace/cse/reference/authorization-tokens

— Google crée le document, en stockant les informations chiffrées reçues par le KACLS.

Lors du chemin inverse, quand un utilisateur souhaite accéder à un document chiffré, les étapes sont les suivantes :

— L'utilisateur accède au Google Workspace de son entreprise et demande l'accès à un document chiffré.

— Le service demande l'authentification de l'utilisateur auprès d'un IdP externe.

— Une fois l'utilisateur authentifié, Google Workspace interroge le KACLS, en lui transmettant la clé *wrappée* correspondant au document, un *token* JWT `authentication` et un *token* JWT `authorization` ainsi qu'une chaîne de caractères JSON `reason`.

— Le KACLS utilise un algorithme de chiffrement authentifié avec données additionnelles pour déchiffrer le tout, effectuer des vérifications sur les *tokens* puis, en cas de succès, renvoyer la DEK.

— Le navigateur va déchiffrer le document en utilisant la DEK déchiffrée reçue par le KACLS, puis l'interface de Google affiche le document voulu.

5.3 Intégration dans notre serveur

5.3.1 Ouverture de notre service vers l'extérieur

Les développements se faisant sur une machine locale en attendant le déploiement sur les serveurs de DuoKey, nous avons dû trouver une solution pour permettre l'accès à notre serveur de l'extérieur afin de permettre de tester l'intégration avec les services de Google. Nous avons utilisé le produit Cloudflare Tunnel [8], anciennement Argo Tunnel [9], qui permet d'exposer le réseau local sur Internet au travers du réseau de Cloudflare.

DuoKey ayant mis à disposition son compte, nous avons pu ouvrir notre application vers l'extérieur après une brève configuration sur la machine de développement. Le service se trouve ainsi accessible via l'URL *https://dke-gcse.duokey.cloud/* .

5.3.2 Chiffrement des données

Nous avons utilisé le chiffrement AEAD, ou chiffrement authentifié avec données additionnelles. Le chiffrement peut se faire avec ou sans ces données additionnelles, mais elles permettent une sécurité supplémentaire en contextualisant le contenu qui doit être chiffré.

Il fonctionne de la manière suivante :

1. Alice et Bob conviennent d'une clé qui sera utilisée lors de leurs échanges.

8. https://www.cloudflare.com/products/tunnel/
9. https://blog.cloudflare.com/argo-tunnel/

2. Alice chiffre un message et authentifie les données additionnelles, qui seront connues de Bob également, à l'aide de la clé choisie. Le tout produit un texte chiffré qui peut être transmis à Bob.
3. Bob reçoit le texte chiffré, et l'utilise de paire avec les données additionnelles authentifiées pour le déchiffrer à l'aide de la clé choisie avec Alice.

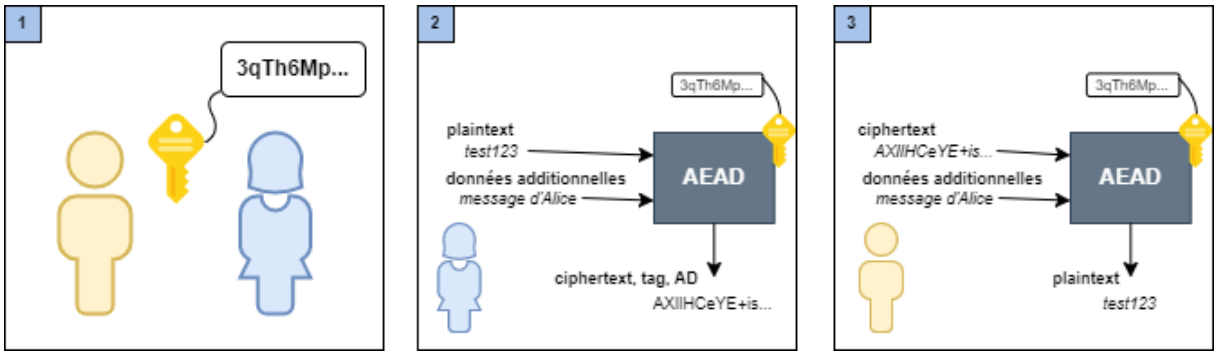

FIGURE 5.3 – *Chiffrement authentifié avec données additionnelles*
Source: *de l'auteur*

En cas d'interception et modification par un tiers malveillant du message lors de la transmission entre Alice et Bob, le déchiffrement échouera et les acteurs seront informés que le message chiffré a été corrompu. Même si la personne possède la clé utilisée pour le chiffrement, sans les données additionnelles elle ne pourra pas déchiffrer le message.

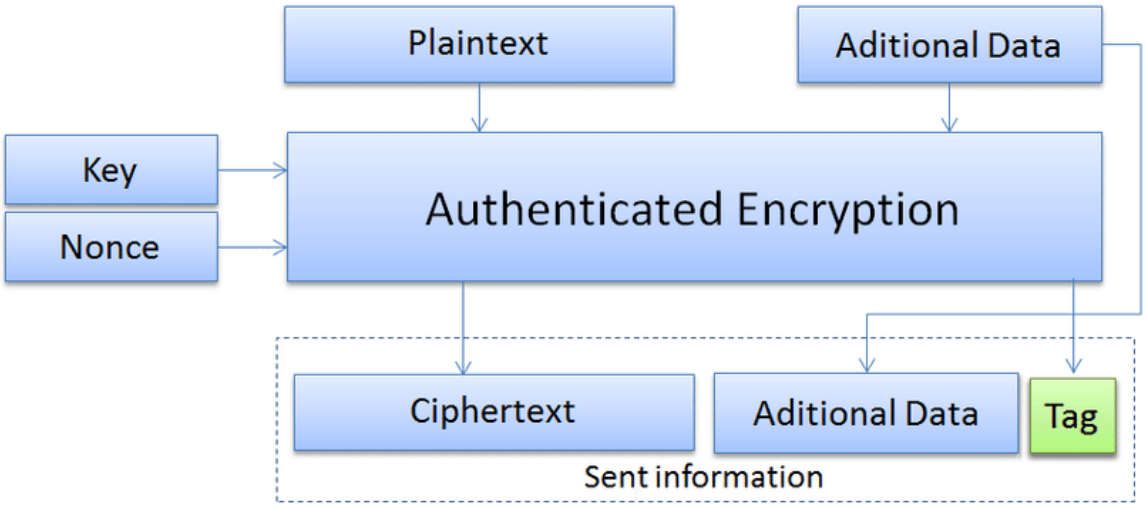

FIGURE 5.4 – *Schéma de fonctionnement de l'AEAD*
Source: *https:// www.researchgate.net/ figure/ Basic-block-design-of-an-AEAD-where-ciphertext-and-authentication-tag-are-produced-by_ fig1_321137002*

Le chiffrement authentifié offre les garanties suivantes [10] :

10. https://developers.google.com/tink/aead

— confidentialité : il n'est pas possible d'obtenir des informations sur la chaîne chiffrée en dehors de sa longueur ;
— authenticité : sans la clé, il n'est pas possible de modifier la chaîne chiffrée sans détection ;
— randomisation : le chiffrement est aléatoire, deux messages identiques ne donneront pas la même chaîne chiffrée ;
— intégrité : le message est protégé de toute modification.

Dans notre cas, nous n'avons pas utilisé pas les données additionnelles car l'API de Google ne permet pas de les transmettre, mais la structure pour leur utilisation est disponible.

5.3.3 Réalisation

La documentation officielle de l'API mentionne plusieurs méthodes à implémenter [11]. Afin d'intégrer la fonctionnalité dans notre serveur Go, nous devons mettre à disposition les *endpoints* suivants :

— `status`, qui est nécessaire à la configuration dans l'administration Google Workspace et qui renvoie un statut sur le KACLS ;
— `wrap`, qui va effectuer le chiffrement avec AEAD et renvoyer une DEK chiffrée ;
— `unwrap`, qui va déchiffrer une DEK.

Les autres méthodes, à savoir `digest`, `takeout_unwrap` et `rewrap` ne sont pas nécessaires au fonctionnement basique de la *client-side encryption* et par conséquent ne seront donc pas implémentées dans le cadre de ce travail. Toutefois, la manière de les appeler n'a pas pu être trouvée dans la documentation actuelle.

Durant les développements, qui peuvent être catégorisés de *proof of concept* (PoC), nous utiliserons une manière non-sécurisée de gérer les clés. Cette manière de procéder n'est pas recommandée dans le cadre d'un projet destiné à la production. Nous utiliserons le *package* `insecurecleartextkeyset` [12] disponible dans la bibliothèque `tink` [13] qui fournit des API cryptographiques sécurisées. Ce package offre la possiblité d'accéder en clair à un *keyset* en lecture et en écriture.

Avant de commencer, nous avons besoin de générer un *keyset* qui sera utilisé par l'AEAD pour chiffrer et déchiffrer les données reçues de Google. Pour ce faire, nous avons employé l'utilitaire `tinkey` [14] qui fait partie de la bibliothèque `tink`. Pour générer ce *keyset*, nous avons utilisé la commande suivante :

```
tinkey create-keyset --key-template AES128_GCM --out-format json --out
↪  aead_keyset.json
```

11. https://developers.google.com/workspace/cse/reference
12. https://github.com/google/tink/tree/master/go/insecurecleartextkeyset
13. https://github.com/google/tink
14. https://github.com/google/tink/blob/master/docs/TINKEY.md

Le contenu de ce fichier JSON est chargé lors du démarrage du serveur.

Dans un second temps, nous allons créer un nouveau fichier *protobuf* qui servira à définir le service ainsi que les messages échangés. Le service `DKEGoogleCSE` définit les méthodes RPC suivantes :

```
// Service pour le Client Side Encryption
service DKEGoogleCSE {
    // Statut du KACLS
    rpc GetStatus(StatusRequest) returns (StatusResponse) {
        option (google.api.http) = {
            get: "/status"
            };
    }
    // Demande pour wrap une clef
    rpc Wrap(WrapRequest) returns (WrapResponse) {
        option (google.api.http) = {
            post: "/wrap"
            body: "*"
            };
    }
    // Demande pour unwrap une clef
    rpc Unwrap(UnwrapRequest) returns (UnwrapResponse) {
        option (google.api.http) = {
            post: "/unwrap"
            body: "*"
            };
    }
}
```

Les différents messages *...Request* et *...Response* sont définis selon la documentation de l'API. Par exemple, le couple *WrapRequest* et *WrapResponse* :

```
// Structure d'une requête wrap
//
// référence : https://developers.google.com/workspace/cse/reference/wrap
message WrapRequest {
    string authentication = 1; // JWT d'authentication
    string authorization = 2; // JWT d'authorization
    string key = 3; // DEK
    string reason = 4; // JSON avec info supplémentaires (facultatif)
}

// Structure d'une réponse wrap
//
// référence : https://developers.google.com/workspace/cse/reference/wrap
message WrapResponse {
    string wrapped_key = 1; // objet binaire encodé en base64 représentant la DEK
    ↪ wrappée
}
```

Deux messages supplémentaires pour la structuration des données à chiffrer avec l'AEAD ont été codés. Ils ont été utilisés afin d'organiser les données qui vont être sérialisées, puis *wrappées*. Nous pourrons plus facilement travailler avec, et lors de l'*unwrap* il suffira de les désérialiser pour les utiliser dans un message *protobuf*.

```
// Informations principales à chiffrer via l'AEAD.
message MessageData {
    // Data Encryption KEY (DEK)
    string key = 1;
    // resource_name récupéré depuis le jwt authorization
    string resource_name = 2;
    // perimeter_id récupéré depuis le jwt authorization
    string perimeter_id = 3;
}

// Informations additionnelles à chiffrer
message AdditionalData {
    // Exemple donnée 1
    string foo = 1;
    // Exemple donnée 2
    string bar = 2;
}
```

Les informations `resource_name` et `perimeter_id` vont être utilisées également lors du *unwrap*, pour effectuer des contrôles supplémentaires. Nous allons vérifier que les valeurs reçues lors du *unwrap* correspondent à celles du *unwrap* déchiffrées.

Les méthodes nécessaires à l'implémentation de la *client-side encryption* ont ensuite été développées. Ceci a été fait dans le fichier `dke_gcse.go`. Il s'articule autour des différents *endpoints* listés précédemment, avec quelques méthodes utilitaires ou factorisant du code utile à plusieurs endroits.

Ci-après se trouvent les étapes nécessaires pour la route `wrap`, les autres sont disponibles dans le code source de notre serveur. La méthode se divise en plusieurs parties :

— récupération du *payload* (DEK, *tokens* `authentication` et `authorization`, `reason`) envoyé par POST,

— validation des *tokens* selon la documentation et récupération des *claims* nécessaires,

— *wrap* du message (DEK, claims `resource_name` et `perimeter_id`) et des données additionnelles,

— journalisation de l'événement,

— renvoi du message *protobuf* contenant la clé *wrappée*, ainsi que les deux *claims* reçus.

Notre méthode qui effectue le *wrap* récupère le *keyset* stocké en clair en mémoire afin de pouvoir effectuer le chiffrement, puis elle sérialise les informations principales ainsi que les données additionnelles. L'ensemble est chiffré via l'AEAD, puis encodé en `base64` afin d'être renvoyé à Google.

Finalement, pour permettre les communications entre les différents acteurs, nous avons du paramétrer les *Cross-Origin Ressource Sharing* (CORS). Cette configuration autorise les appels à des ressources hébergées sur un domaine différent de celui de la requête initiale.

```
// Paramètres CORS pour Google CSE
c := cors.New(cors.Options{
    AllowedOrigins: []string{"https://admin.google.com",
↪   "https://client-side-encryption.google.com"},
    AllowedHeaders: []string{"*"},
    AllowedMethods: []string{"POST", "GET"},
})
```

Le serveur peut ensuite démarrer, et l'utilisaeteur peut effectuer la création d'un document chiffré depuis le Google Workspace de son organisation.

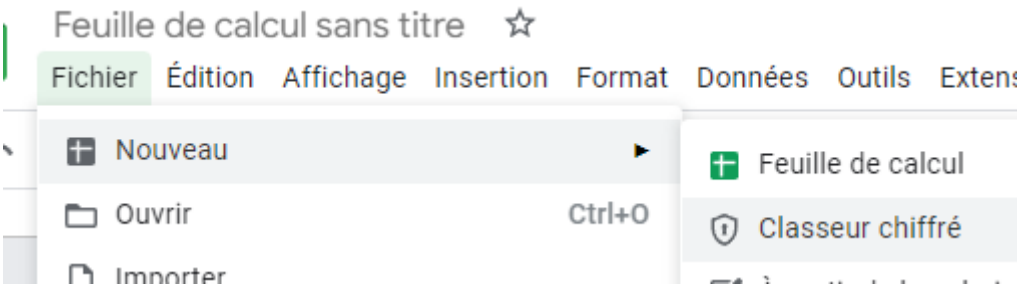

FIGURE 5.5 – *Création d'un nouveau classeur chiffré*
Source: *de l'auteur*

Lors de la première itération, Google demande de s'authentifier auprès de l'IdP :

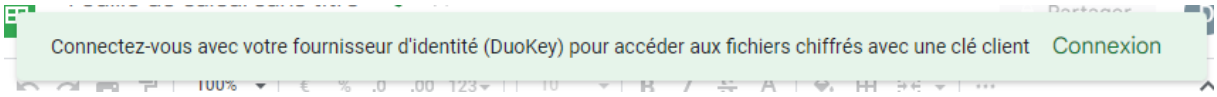

FIGURE 5.6 – *Demande d'authentification auprès de l'identity provider*
Source: *de l'auteur*

En utilisant la console développeur du navigateur, les échanges effectués peuvent être observés, notamment l'appel à la méthode `wrap`. Dans les deux captures ci-dessous, nous avons les informations *payload* envoyé au serveur. Dans la seconde nous avons la clé *wrappée* renvoyée par le KACLS.

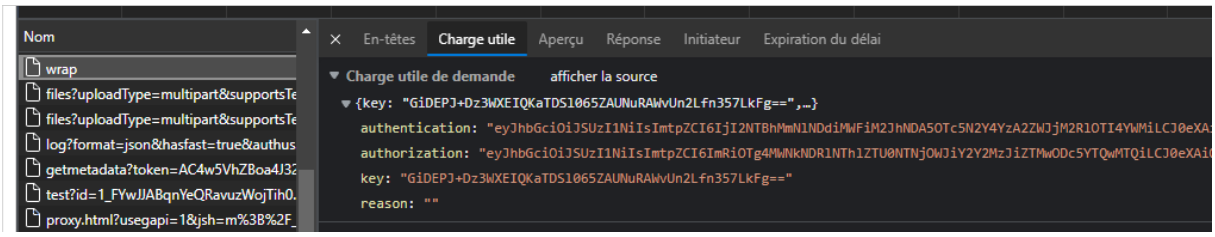

FIGURE 5.7 – *Payload envoyé par Google lors du* `wrap`
Source: *de l'auteur*

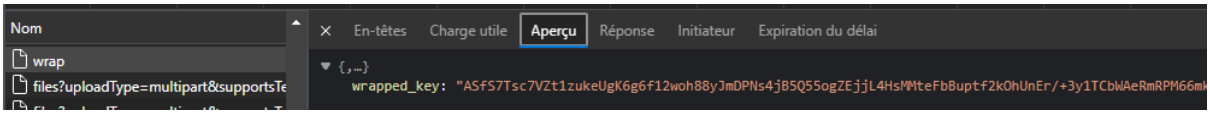

FIGURE 5.8 – *Clé wrappée renvoyée par notre KACLS lors du* `wrap`
Source: *de l'auteur*

Une fois le processus terminé, le document est indiqué comme chiffré par l'organisation du Google Workspace.

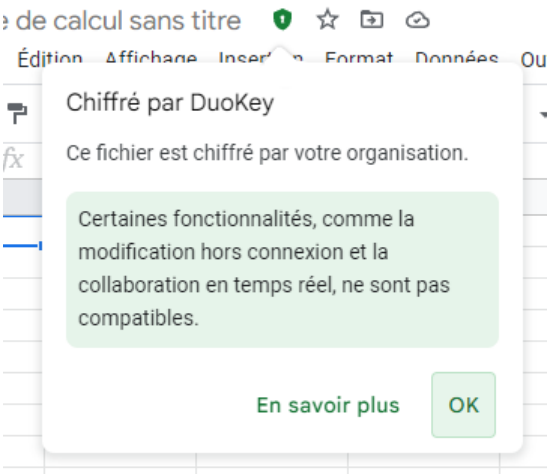

FIGURE 5.9 – *Confirmation du chiffrement du document*
Source: *de l'auteur*

Après enregistrement, le fichier chiffré se retrouve dans la liste des documents du Google Drive :

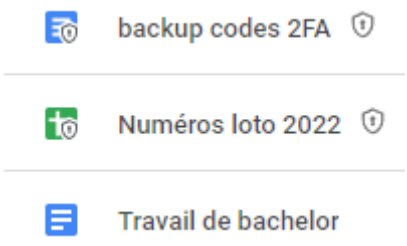

FIGURE 5.10 – *Liste des documents de l'utilisateur*
Source: *de l'auteur*

6 Intégration du serveur dans une image Docker

6.1 Introduction

La dernière tâche à réaliser dans le cadre de ce travail est de mettre à disposition le serveur via un conteneur Docker. Nous allons commencer par une présentation de Docker et expliquer ce qui est possible de faire à l'aide de cette technologie. Nous présenterons ensuite ce qui a été réalisé dans le cadre de ce projet, et comment l'application développée a été intégrée dans un conteneur.

6.2 Présentation de Docker

Docker est une plateforme proposée par la société du même nom permettant de développer, livrer et exécuter des applications. La fonctionnalité principale offerte est la possiblité d'isoler complètement de son hôte l'environnement d'exécution d'une application. Cet environnement sécurisé est appelé conteneur.

Un conteneur est relativement indépendant de son hôte, et dispose en interne de l'ensemble des éléments nécessaires à son bon fonctionnement. Il se base sur une image existante, publiée sur un *registry*, ou peut être construit selon les instructions d'un *Dockerfile*. Sur la machine hôte, le Docker *daemon* est responsable de plusieurs tâches, par exemple de l'orchestration des *containers* ou de la gestion des images de référence.

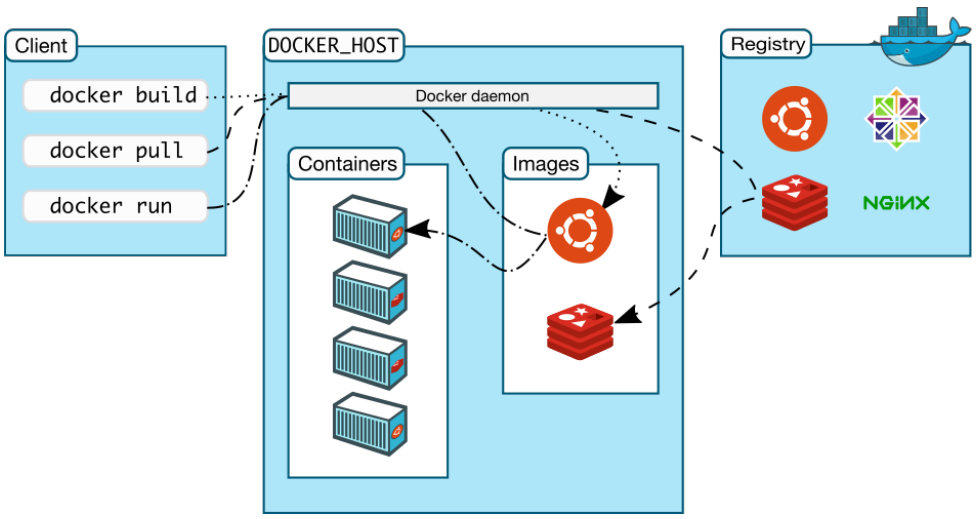

FIGURE 6.1 – *Architecture Docker*
Source: *de l'auteur à partir de https://docs.docker.com/get-started/overview/*

Comme expliqué précédemment, il est possible de construire une image à partir des instructions d'un *Dockerfile*. Ce fichier, rédigé en YAML, représente les différentes étapes nécessaires à la construction d'une image qui sera utilisée par un conteneur. Le code ci-dessous représente les étapes nécessaires à la construction d'une image contenant une application *Node.js*, reprises de la documentation officielle Docker [1].

```
# syntax=docker/dockerfile:1
FROM node:12-alpine
RUN apk add --no-cache python2 g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Il est ensuite possible de construire notre image à partir de ces instructions, à l'aide de la commande `docker build`.

```
$ docker build -t image-exemple .
```

Une fois l'image générée, elle permet de créer un conteneur s'y référant via la commande `docker run`.

```
$ docker build -t image-exemple .
```

Dans le *Dockerfile* d'exemple, l'image expose le port 3000 avec l'instruction `EXPOSE`. Il est donc nécessaire d'indiquer quel port de la machine hôte communiquera avec le port 3000 du conteneur.

6.3 Création de notre image

Selon le principe vu précédemment, nous allons créer un *Dockerfile* à la racine du projet afin de décrire les étapes nécessaires à la génération de l'image. Nous exploitons le principe dit de *build multi-stage*, ce qui permet de réduire drastiquement la taille de l'image finale. Une image de référence comprenant les outils nécessaire au *build* de l'application est utilisée, et une autre plus légère pour son exécution. Dans notre cas, l'image pour le *build* est `golang:1.17.6-alpine` [2], et celle pour l'exécution `scratch` [3]. Une fois compilée, notre application Go étant *linkée* statiquement nous n'avons pas besoin de bibliothèques externes et il est possible de partir d'une image de référence presque vide. De ce fait, une image finale d'environ 36 Mo est construite.

1. https://docs.docker.com/get-started/02_our_app/
2. https://hub.docker.com/_/golang
3. https://hub.docker.com/_/scratch

Le *Dockerfile* complet est disponible dans les annexes. Nous récapitulerons ici les étapes effectuées. Premièrement, dans l'étape `build` :

— création d'un utilisateur sans privlièges *root* qui servira à exécuter notre application,

— installation des *root CA*, afin de permettre la communication sécurisée lors de la récupération des JWKS,

— copie du `go.mod` et `go.sum` qui seront nécessaires lors du *build*,

— téléchargement des dépendances,

— copie des fichiers nécessaires au *build*,

— *build* de l'application.

Ci-dessous se trouvent les activités de la seconde étapes, en utilisant l'image `scratch` :

— récupération des fichiers `/etc/passwd` et `/etc/group` générés précédemment,

— copie des fichiers nécessaires à Swagger UI[4],

— copie de l'exécutable du serveur,

— copie des *root CA* depuis le *stage* `build`,

— exposition des ports nécessaires aux communications avec gRPC et *grpc-gateway*,

— création des volumes qui vont accueillir les certificats et *keyset* de l'utilisateur,

— spécification de l'instruction d'exécution de l'image.

Une fois les étapes de construction de l'image définies, il est possible de la générer à l'aide de la commande `docker build`[5]. Le paramètre `-t` permet de configurer un nom et un *tag* optionnel sur notre image.

```
$ docker build -t grpc-rest-server:latest .
```

Nous pouvons ensuite afficher les images disponibles à l'aide de la commande `docker image ls`, ce qui nous donne le résultat suivant :

```
$ docker image ls
REPOSITORY         TAG         IMAGE ID       CREATED         SIZE
grpc-rest-server   go-1.18.3   4b9aa29103ea   38 hours ago   35.8MB
grpc-rest-server   latest      0f4dba49247c   38 hours ago   35.6MB
```

Ici deux images `grpc-rest-server` sont visibles, une avec le *tag* `latest` qui est le *tag* saisi par défaut, et une autre avec une version différente de Go utilisée pour tester le code. Finalement, nous pouvons le conteneur avec notre image à l'aide de la commande `docker run`[6] :

4. uniquement dans le cadre de notre PoC
5. https://docs.docker.com/engine/reference/commandline/build/
6. https://docs.docker.com/engine/reference/commandline/run/

```
$ docker run -v "$(pwd)"/keyset:/keyset -v "$(pwd)"/cert:/cert -p
↪  8485:8485 -p 8486:8486 -d --name grpc-server-test
↪  grpc-rest-server:latest run -s TLS_NONE
```

Les deux paramètres `-v` qui permet d'effectuer un montage entre le *file system* hôte et celui du conteneur. Dans la même logique, les instructions `-p` publient un port de l'hôte depuis le conteneur. L'option `-d` permet l'exécution en mode *detached*, c'est-à-dire en tâche de fond. Le paramètre `-name` est explicite, il permet de nommer le conteneur créé. Nous indiquons ensuite quelle image utiliser, puis l'instruction `run` est un paramètre propre à notre serveur qui lui indique de démarrer. La dernière option `-s TLS_NONE` est également un paramètre de notre application, nous lui indiquons de démarrer sans configuration TLS. Un *container ID* est renvoyé en cas de succès.

Avec la commande `docker ps` nous pouvons voir que la création s'est bien déroulée correctement [7] :

```
$ docker ps
CONTAINER ID   IMAGE                    COMMAND                  STATUS         PORTS                     NAMES
9252d1fa8941   grpc-rest-server:latest   "./server run -s TLS... " Up 24 seconds   8485-8486->8485-8486/tcp   grpc-server-test
```

La commande `docker logs <container id>` permet d'afficher la sortie de la console du conteneur :

```
$ docker logs 9252d1fa8941
2022-07-08T10:04:28Z    INFO    setting up server
2022-07-08T10:04:28Z    INFO    serving gRPC on port 8486
2022-07-08T10:04:28Z    INFO    swagger configuration found
2022-07-08T10:04:28Z    INFO    keyset info loaded and stored in memory
2022-07-08T10:04:28Z    INFO    serving gRPC-gateway without SSL on port 8485
```

Nous pouvons atteindre l'interface *Swagger UI* de l'application, ou accéder à un document chiffré.

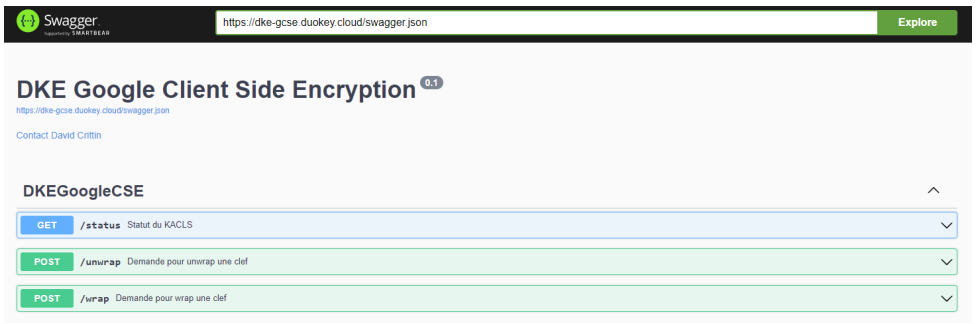

FIGURE 6.2 – *Swagger UI de notre application intégrée à un conteneur*
Source: *de l'auteur*

7. certaines colonnes ont été omises pour faciliter la lecture

7 | Conclusion

La prise en charge de ce projet nous a permis de découvrir le langage Go, que nous n'avions pas eu la chance d'appréhender durant le cursus à la HES-SO. Nous avons ainsi pu le prendre en main, apprendre à l'utiliser et grâce au sujet de ce travail nous avons eu l'opportunité de réaliser un projet concret qui nous l'espérons sera utilisé dans le futur. Mettre à disposition une solution quasiment prête à la production en n'ayant aucune connaissance du langage de programmation au départ, tout cela en moins de 6 mois et en prenant en compte nos agendas chargés par la vie professionnelle, privée et estudiantine, fut un défi remarquable.

Nous avons également eu la chance de pouvoir collaborer avec une entreprise durant ce travail. Notre interlocuteur au sein de DuoKey nous a permis de comprendre la tâche à réaliser et grâce à leur collaboration proactive nous avons pu mener à bien ce projet. Cela nous a permis de nous rendre compte des standards qui peuvent être attendus au sein d'une entreprise.

Difficultés rencontrées

Durant la réalisation de ce projet, nous avons rencontrés un certain nombre de problèmes variés. Fort heureusement, aucun d'entre eux ne nous a bloqués durant trop longtemps, ou a demandé la remise en question un élément du cahier des charges du projet. La grande majorité des difficultés rencontrées a pu être surmontée à l'aide de solutions proposées de la communauté, ou en demandant du support à Jean-Luc.

Nous releverons en particulier un problème qui a failli mettre en péril l'utilisation de la solution proposée par Google. En effet, le produit étant encore en *closed beta*, l'interface d'administration peut encore laisser à désirer. DuoKey s'est retrouvée bloqué lorsqu'il a fallu mettre à jour l'adresse du KACLS, le champ n'étant pas éditable. Il s'en est suivi de nombreux échanges avec le support Google Workspace qui après quasiment un mois a mis à jour le champ avec la nouvelle valeur, nous permettant de tester l'intégration des développements plus récents. Sans ce déblocage, nous aurions orienté nos efforts sur la solution de Microsoft.

Axes d'amélioration

Bien entendu, il reste des axes d'amélioration pour ce travail, tels que l'utilisation d'un KMS externe plutôt qu'un *keyset* non-sécurisé en mémoire. En effet, dans le cadre d'un projet destiné à la production, l'intégration d'un *keyset* non-sécurisé et en local ne satisfait pas les principes de sécurité attendus pour ce type d'applications.

Comme indiqué lors de l'intégration de la solution de *client-side encryption* de Google (sous-section 5.3.3), nous avons implémenté uniquement les *endpoints* nécessaires au fonctionnement de notre projet. Il serait donc possible d'intégrer les trois mentionnés dans la documentation officielle qui n'ont pas été développés.

Dans le but de comparer les solutions, l'implémentation de celle de Microsoft pourrait être une piste intéressante. Leur produit Double Key Encryption faisait partie des *use cases* envisageables mentionnés dans le cahier des charges de ce travail.

Finalement, nous avons également intégré un certains nombres de tests unitaires. Ceux-ci pourraient être complétés. Nous avons pu réaliser une couverture limitée de notre code, le point concernant les tests unitaires ayant été abordé sur le tard, d'un commun accord avec le mandant.

I | Configuration des projets Azure et Google

Nous avons configuré notre projet dans l'Azure Active Directory selon la procédure mentionnée dans la bibliographie [1]. Celle-ci nous a permis d'obtenir des *tokens* JWT et de les valider au travers de l'application.

I.1 Projet dans Azure Active Directory

Cette section montre l'écran de vue d'ensemble du projet dans la configuration Azure. Les informations utiles sont les différents identifiants visibles sur la capture d'écran : *application id*, *object id* et *directory id*. Ces données seront nécessaires lors de la demande de *token*.

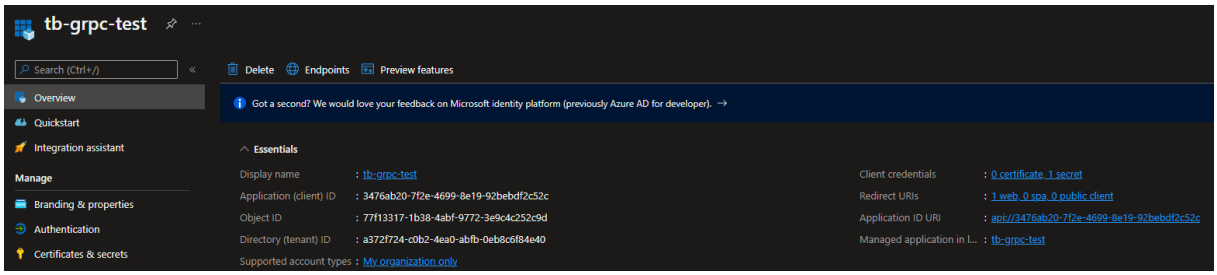

FIGURE I.1 – *Vue d'ensemble du projet*
Source: *de l'auteur à partir de portal.azure.com*

Dans la capture ci-dessous, nous retrouvons le *secret id* ainsi que sa valeur. Cette dernière est également nécessaire à l'obtention de *token* JWT.

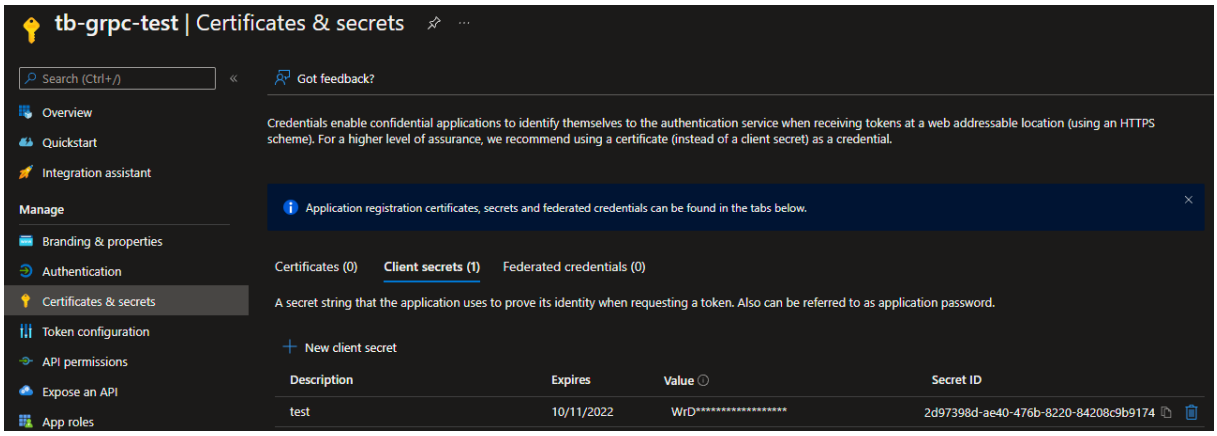

FIGURE I.2 – *Certificats et secrets*
Source: *de l'auteur à partir de portal.azure.com*

Nous avons dû configurer des permissions pour permettre aux utilisateurs d'accéder à l'application. La capture ci-dessous montre la permission qui a été définie dans le projet.

1. https://blog.jonathanchannon.com/2022-01-29-azuread-golang/

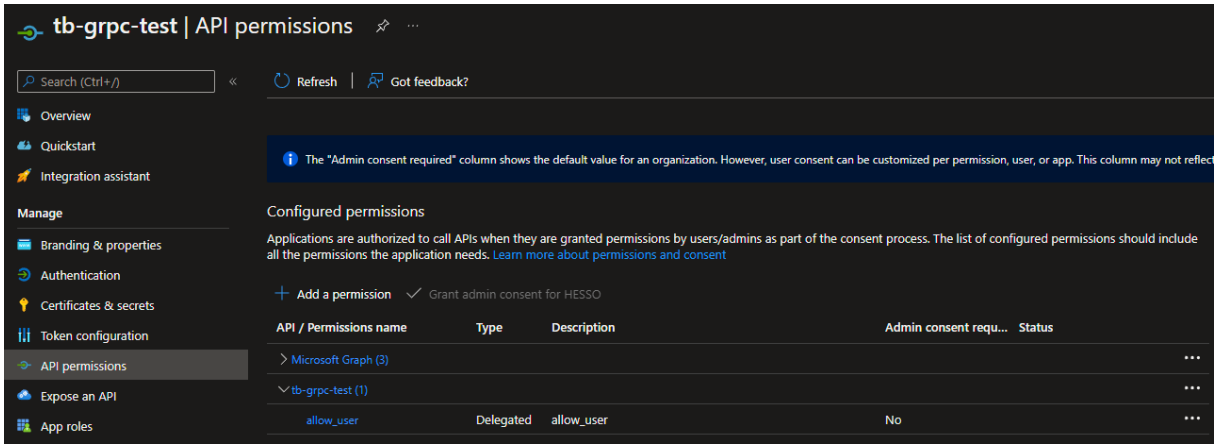

FIGURE I.3 – *Permissions de l'API*
Source: *de l'auteur à partir de portal.azure.com*

Finalement, les permissions ci-dessus sont affectées à un *scope*, selon la capture ci-dessous. Cela permet de définir des restrictions d'accès aux données ou aux fonctionnalités exposées par l'application. Nous devons également indiquer à l'API quelle application est autorisée.

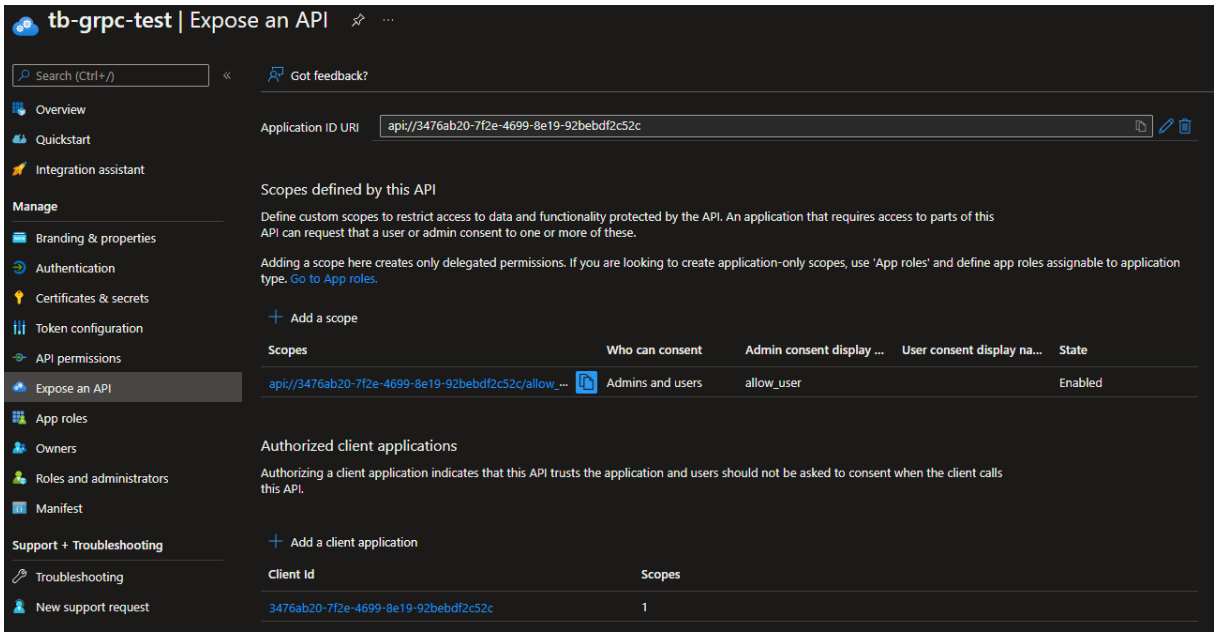

FIGURE I.4 – *Scopes et applications clientes*
Source: *de l'auteur à partir de portal.azure.com*

I.2 Projet dans Google Cloud Platform

Dans cette section, nous présentons la configuration effectuée dans le projet au sein de Google Cloud Platform. Tout d'abord le projet a été créé et des identifiants clients ont été configurés comme visible dans la capture d'écran ci-dessous.

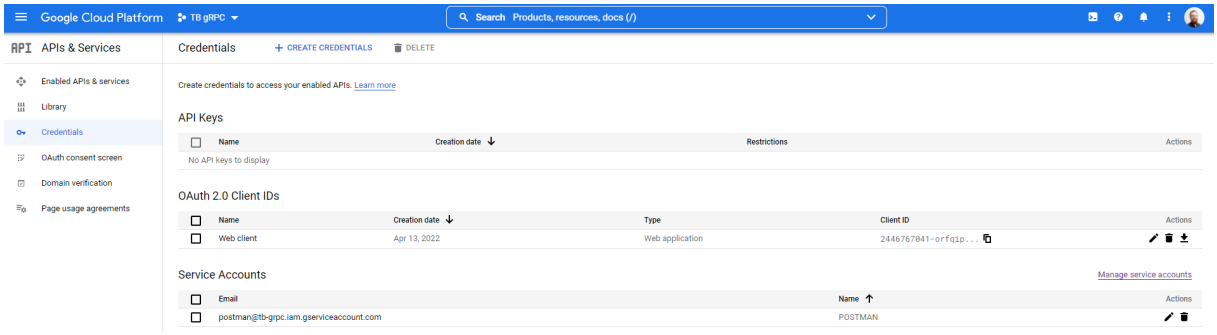

FIGURE I.5 – *Credentials de l'API*
Source: *de l'auteur à partir de console.cloud.google.com*

Nous retrouvons des informations similaires au projet Azure, avec notamment des identifiants tels que *client id* ainsi que *client secret*, ce dernier ayant été volontairement partiellement obfusqué.

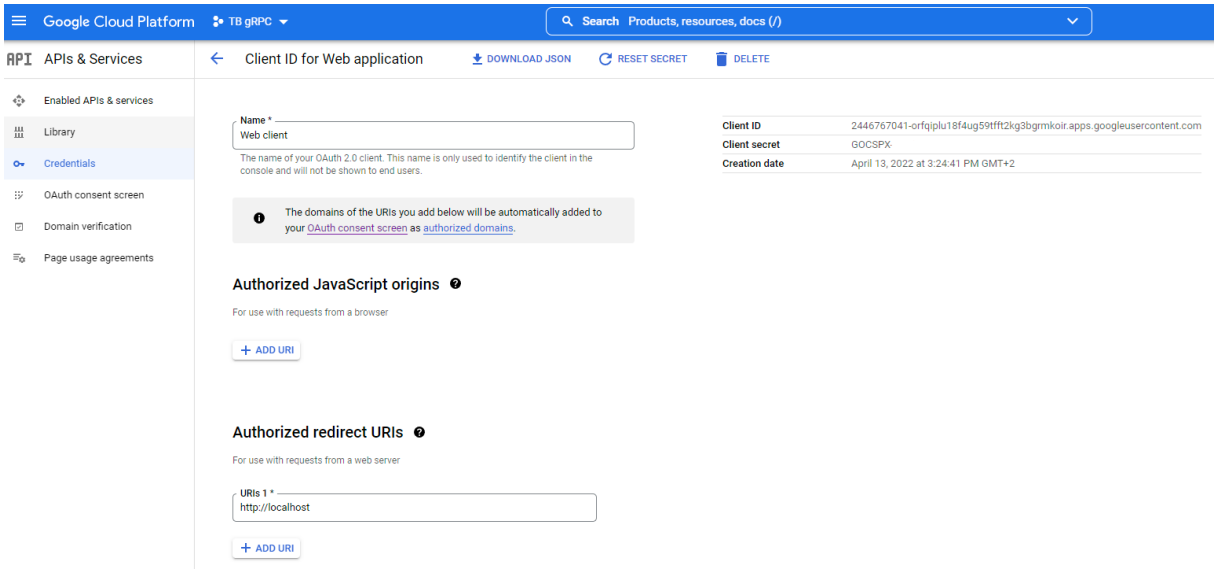

FIGURE I.6 – *Client OAuth 2.0*
Source: *de l'auteur à partir de console.cloud.google.com*

II | Script de génération des clés et certificats

```
rm *.pem

# 1. Generate CA's private key and self-signed certificate
echo "Generate CA's private key and self-signed certificate"
openssl req -x509 -newkey rsa:4096 -days 365 -nodes -keyout
↪  ca-key.pem -out ca-cert.pem -subj
↪  "/C=CH/ST=Valais/L=Martigny/O=HEVs/OU=Students/CN=*.tbgrpc.ch/ema
↪  ilAddress=david.crittin@students.hevs.ch"

# 2. Generate web server's private key and certificate signing
↪  request (CSR)
echo "Generate web server's private key and certificate signing
↪  request (CSR)"
openssl req -newkey rsa:4096 -nodes -keyout server-key.pem -out
↪  server-req.pem -subj "/C=CH/ST=Valais/L=Martigny/O=HEVs/OU=Studen
↪  ts/CN=*.tbgrpc.ch/emailAddress=david.crittin@students.hevs.ch"

# 3. Use CA's private key to sign web server's CSR and get back the
↪  signed certificate
echo "Use CA's private key to sign web server's CSR and get back the
↪  signed certificate"
openssl x509 -req -in server-req.pem -days 60 -CA ca-cert.pem -CAkey
↪  ca-key.pem -CAcreateserial -out server-cert.pem -extfile
↪  server-ext.cnf

#echo "Server's signed certificate"
#openssl x509 -in server-cert.pem -noout -text

echo "Generate server-cert.key"
openssl pkey -in server-key.pem -out server-cert.key
echo "Generate server-cert.crt"
openssl x509 -in server-cert.pem -out server-cert.crt

# 4. Generate client's private key and certificate signing request
↪  (CSR)
echo "Generate client's private key and certificate signing request
↪  (CSR)"
openssl req -newkey rsa:4096 -nodes -keyout client-key.pem -out
↪  client-req.pem -subj "/C=CH/ST=Valais/L=Martigny/O=HEVs/OU=Studen
↪  ts/CN=*.tbgrpc.ch/emailAddress=david.crittin@students.hevs.ch"
```

```
    # 5. Use CA's private key to sign client's CSR and get back the
    ↪  signed certificate
    echo "Use CA's private key to sign client's CSR and get back the
    ↪  signed certificate"
    openssl x509 -req -in client-req.pem -days 60 -CA ca-cert.pem -CAkey
    ↪  ca-key.pem -CAcreateserial -out client-cert.pem -extfile
    ↪  client-ext.cnf
```

III | Fichier Dockerfile pour la génération d'une image Docker

```
# syntax=docker/dockerfile:1

#########
# BUILD #
#########
# Utilisation de l'image golang officielle, version 1.17.6
FROM golang:1.17.6-alpine as build

# Répertoire de travail
WORKDIR /app

# Création de l'utilisateur qui exécutera l'application
# selon procédure => https://stackoverflow.com/a/55757473/12429735RUN
ENV USER=appuser
ENV UID=10001

# Création de l'utilisateur sans privilèges root
RUN adduser --disabled-password --gecos "" --home "/nonexistent" --shell
 ↪  "/sbin/nologin" --no-create-home --uid "${UID}" "${USER}"

# Installation des root CAs
RUN apk add --no-cache ca-certificates

# Copie des fichiers nécessaires au "go build"
COPY go.mod ./
COPY go.sum ./

# Téléchargement des fichiers nécessaires aux imports
RUN go mod download

# Copie depuis notre projet des fichiers de notre application nécessaires
 ↪  au build
COPY cmd/ ./cmd/
COPY internal/ ./internal/
COPY *.go ./

# Build de notre application
RUN export CGO_ENABLED=0 && go build /app/cmd/server

########
# EXEC #
```

```
########
# Utilisation de l'image "scratch" pour avoir le strict nécessaire et une image légère
FROM scratch

# Répertoire de travail
WORKDIR /

# Récupération des éléments nécessaires depuis l'étape "build"
# Fichiers nécessaires pour l'utilisateur sans privilèges root
COPY --from=build /etc/passwd /etc/passwd
COPY --from=build /etc/group /etc/group

# Fichiers swagger-ui
COPY --from=build /app/internal/api/swagger /internal/api/swagger

# Exécutable du serveur
COPY --from=build /app/server /server

# Root CAs
COPY --from=build /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/

# Utilisation du user créé lors du build
USER appuser:appuser

# Exposition des ports nécessaires (grpc et grpc-gateway)
EXPOSE 8485
EXPOSE 8486

# Volumes nécessaires pour les fichiers pour le TLS et pour le keyset en mémoire
VOLUME [ "/cert" ]
VOLUME [ "/keyset" ]

# Ce qui s'exécute avec notre image
ENTRYPOINT [ "./server" ]
```

IV | Planification des *epics*

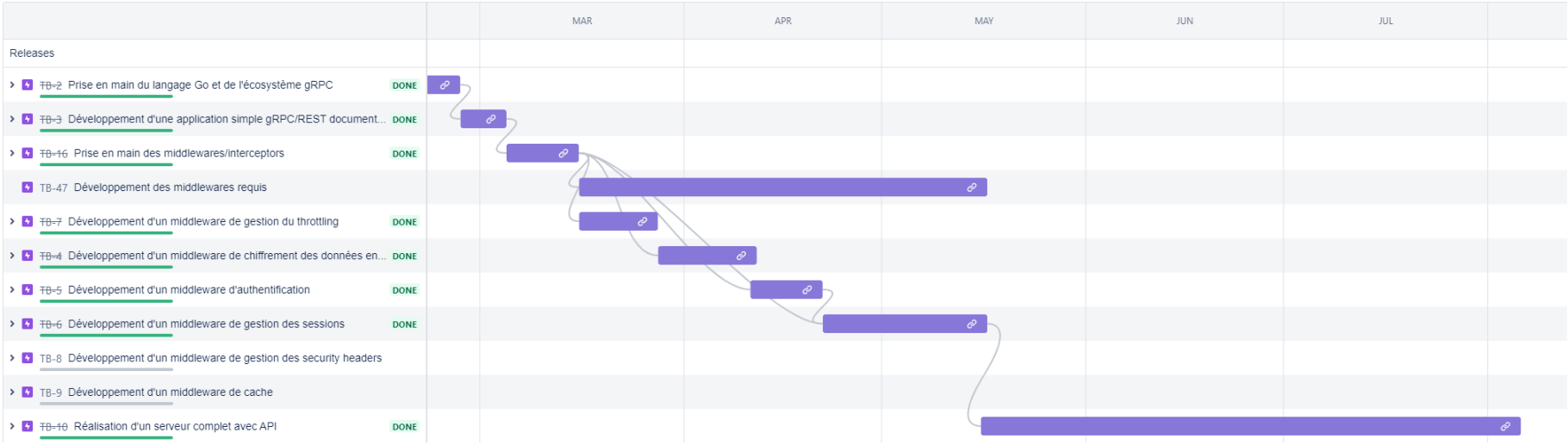

FIGURE IV.1 – *Annexes - Planification des epics*
Source: *de l'auteur*

Références

A complete overview of SSL/TLS and its cryptographic system. (2021). Récupérée le 10 avril 2022, à partir de https://dev.to/techschoolguru/a-complete-overview-of-ssl-tls-and-its-cryptographic-system-36pd

Configuring CORS for Go (Golang). (2021). Récupérée le 20 juin 2022, à partir de https://www.stackhawk.com/blog/configuring-cors-for-go/

Customizing your gateway. (2021). Récupérée le 26 février 2022, à partir de https://grpc-ecosystem.github.io/grpc-gateway/docs/mapping/customizing_your_gateway/

Docker overview. (2022). Récupérée le 6 juillet 2022, à partir de https://docs.docker.com/get-started/overview/

Docker overview. (2022). Récupérée le 6 juillet 2022, à partir de https://docs.docker.com/storage/bind-mounts/

Generating Swagger Docs From Go. (2021). Récupérée le 26 février 2022, à partir de https://ldej.nl/post/generating-swagger-docs-from-go/

Get Azure AD tokens by using the Microsoft Authentication Library. (2022). Récupérée le 13 avril 2022, à partir de https://docs.microsoft.com/en-us/azure/databricks/dev-tools/api/latest/aad/app-aad-token

Go quick start. (2022). Récupérée le 24 février 2022, à partir de https://grpc.io/docs/languages/go/quickstart/

GoLang api logging with correlation ID. (2019). Récupérée le 12 mars 2022, à partir de https://dejanvasic.wordpress.com/2019/11/01/golang-logging-with-correlation-id/

Google Authentication with Postman. (2020). Récupérée le 14 avril 2022, à partir de https://medium.com/kinandcartacreated/google-authentication-with-postman-12943b63e76a

Google Workspace CSE API Reference. (2021). Récupérée le 1 juin 2022, à partir de https://developers.google.com/workspace/cse/reference

Google Workspace delivers new levels of trusted collaboration for a hybrid work world. (2021). Récupérée le 31 mai 2022, à partir de https://cloud.google.com/blog/products/workspace/new-google-workspace-security-features

How to secure gRPC connection with SSL/TLS in Go. (2021). Récupérée le 12 mars 2022, à partir de https://dev.to/techschoolguru/how-to-secure-grpc-connection-with-ssl-tls-in-go-4ph

interceptor in gRPC. (2021). Récupérée le 2 avril 2022, à partir de https://programmingpercy.tech/blog/grpc-interceptor/

Introduction to JSON Web Tokens. (s. d.). Récupérée le 3 mai 2022, à partir de https://jwt.io/introduction

Language Guide (proto3). (2022). Récupérée le 2 mars 2022, à partir de https://developers.google.com/protocol-buffers/docs/proto3

OAuth 2.0 Scopes for Google APIs. (2021). Récupérée le 14 avril 2022, à partir de https://developers.google.com/identity/protocols/oauth2/scopes

OpenAPI Swagger. (2022). Récupérée le 26 février 2022, à partir de https://go-kratos.dev/en/docs/guide/openapi/

Securing a Go Microservice with JWT. (2021). Récupérée le 3 mai 2022, à partir de https://fusionauth.io/blog/2021/02/18/securing-golang-microservice

Tink for Go HOW-TO. (2021). Récupérée le 13 juin 2022, à partir de https://github.com/google/tink/blob/master/docs/GOLANG-HOWTO.md

Tracing gRPC with Istio. (2018). Récupérée le 12 mars 2022, à partir de https://aspenmesh.io/tracing-grpc-with-istio/

Tutorial : Get started with Go. (s. d.). Récupérée le 18 février 2022, à partir de https://go.dev/doc/tutorial/getting-started

USING AZURE AD AUTHENTICATION FOR A GO API. (2022). Récupérée le 13 avril 2022, à partir de https://blog.jonathanchannon.com/2022-01-29-azuread-golang/

Using TLS in grpc-gateway. (2018). Récupérée le 12 mars 2022, à partir de https://gist.github.com/mayankcpdixit/d2260431de5d2a1b2569bace1716f2af

Utiliser JWT pour authentifier les utilisateurs. (2021). Récupérée le 14 avril 2022, à partir de https://cloud.google.com/api-gateway/docs/authenticating-users-jwt

Glossaire

Application Programming Interface Façade proposée par un logiciel afin d'offrir des services à d'autres programmes informatiques. ii

Base64 Codage d'une information en utilisant 64 caractères. 34, 46

Chunk Fragment d'information. 16

Cross-Origin Ressource Sharing Mécanisme basé sur les *headers* HTTP permettant à un serveur d'obtenir des ressources hébergées sur un domaine différent de celui de la requête initiale. 46

Data Encryption Key Type de clé destinée à chiffrer et déchiffrer des données. vii, 40

Epic En développement Agile, regroupe un ensemble de tâches à effectuer ayant un but commun. viii, 1, 61

Identity Provider Service qui s'occupe de stocker, de gérer et de contrôler l'identité d'un utilisateur. 40

Interceptor Implémentation des *middlewares* en Go. ii, vii, 16, 22, *voir* middleware

Key Access Control List Service Service de gestion des clés d'accès, qui permet à un utilisateur de contrôler les clés utilisées pour chiffrer ses données avant de les transmettre à un fournisseur *cloud*. 40

Middleware Composant informatique servant à connecter des applications entre elles. En Go, ceci est réalisé grâce aux *interceptors*. ii, vii, 1, 16, 17, 21, 39

Pool Ensemble de ressources prêtes à l'utilisation stockées en mémoire. 18

Product Backlog En développement Agile, représente une liste priorisée des fonctionnalités voulues dans un projet informatique. 1

Proof of concept Démonstrateur, réalisation ayant pour but de démontrer la faisabilité d'un projet. 43

Story En développement Agile, représente une petite partie d'une fonctionnalité, définie dans un langage orienté utilisateur. 1, 64

Task En développement Agile, représente le découpage du travail à effectuer pour implémenter une story. 1

Timestamp Représentation du temps écoulé en secondes depuis le 01.01.1970 UTC. 35

YAML YAML est un language de programmation permettant de sérialiser des données. Il est souvent utilisé pour décrire des fichiers de configuration, comme les *Dockerfile* ou les instructions d'intégration continue Gitlab. 49

Informations sur ce travail

Informations de contact

Auteur : David Crittin

HES-SO Valais-Wallis

E-mail : *david.crittin@students.hevs.ch*

Déclaration sur l'honneur

Je déclare, par ce document, que j'ai effectué le travail de bachelor ci-annexé seul, sans autre aide que celles dûment signalées dans les références, et que je n'ai utilisé que les sources expressément mentionnées. Je ne donnerai aucune copie de ce rapport à un tiers sans l'autorisation conjointe du RF et du professeur chargé du suivi du travail de bachelor, à l'exception des personnes qui m'ont fourni les principales informations nécessaires à la rédaction de ce travail.

Lieu, date : _____

Signature : _____