GoGuard

Seminarski rad iz predmeta Dizajn Programskih Jezika

Stefan Drljević

12. avgust 2025.

Sadržaj

1. Uvod

1.1 Cilj rada

Cilj ovog semiraskog rada jeste da prikaže dizajn i implementaciju srevera za licenciranje razvijenom u programskom jeziku Go Lang.
On obezbeđuje mehanizme za generisanje i validaciju licenci vezanih za neki hardverski identifikator (HWID), sigurnu razmenu podataka korišćenjem kriptografije (RSA i AES-256-GCM) i takođe zaštitu od prekomernih zahteva putem rate-limiting-a.
Kroz rad biće analizirane ključne tehničke odluke, arhitektura sistema i način na koji odabrani programski jezik Go Lang utiče na strukturu i efikasnost rešenja.

1.2 Kontekst projekta

GoGuard je razvijen kao odgovor na potrebu za jednostavnim, brzim i sigurnim sistemom licenciranja softvera, koji je otporan na neovlašćenu upotrebu i pokušaje zaobilaženja zaštite.
Projekat je realizovan u okviru predmeta ***Dizajn programskih jezika*** sa ciljem da pokaže kako karakteristike Go jezika — kao što su ugrađena podrška za konkurentnost, jednostavan sistem upravljanja memorijom, bogata standardna biblioteka i modulacija koda — mogu biti iskorišćene za izgradnju pouzdane serverske aplikacije.
Sistem se oslanja na Gin framework za HTTP rutiranje, standardni `crypto` paket za kriptografiju, `sync.RWMutex` za sinhronizaciju pristupa deljenim resursima (sa optimizacijom za višestruka čitanja), kao i integraciju sa Microsoft SQL Server bazom podataka za trajno skladištenje licenci. ***Rate limiting*** je implementiran radi zaštite od zloupotrebe i očuvanja performansi servera.

2. Opis problema

2.1 Piraterija i neovlašćen pristup

Softverska piraterija i neovlašćeni pristup predstavljaju značajne izazove u oblasti zaštite intelektualne svojine. Uređaji (Kontroleri) zasnovani na **Raspberry Pi** platformi i ostali ugrađeni sistemi često koriste softver koji je smešten na fizičkim medijima poput SD kartica. Ovo omogućava da se jednostavno kopira softver kloniranjem cele SD kartice što može dovesti do nelegalne distribucije i generalne zloupotrebe softvera.

2.2 Potreba za HWID licenciranjem

Da bi se sprečilo nelegalno korišćenje i distribucija softvera, neophodno je implementirati mehanizme vezivanja licenci za konkretni hardver. **Hardverski ID** (HWID) je jedinstveni identifikator koji se dodeljuje hardverskim komponentama. Možemo ga zamisliti kao digitalni „otisak prsta“ našeg uređaja, koji pomaže da se razlikuje od drugih uređaja. On uglavnom se generiše na osnovu specifičnih karakteristika njegovih hardverskih komponenti. To uključuje procesor, memoriju, matičnu ploču, mrežni adapter i druge ključne delove. Svaka hardverska komponenta ima svoj jedinstveni HWID.
U kontekstu uređaja zasnovanih na Linux platformi, poput Raspberry Pi, čak i ako neko iskopira SD karticu sa softverom i pokuša da je iskoristi na drugom uređaju, softver neće raditi bez validne licence vezane za HWID tog uređaja.

2.3 Sigurna komunikacija

Zbog osetljivosti informacija koje se razmenjuju izmedju klijenta i servera (HWID i licenca) potrebno obezbediti sigurnu komunikaciju tako da se oteza svaki moguci pokusaj presretanja poruka i dolazenja do lakih informacija tim putem.
Hibridna kriptografija koja kombinuje asimetrične (**RSA**) i simetrične (**AES-256-GCM**) metode enkripcije, omogucava nam da komunikacija pored toga sto je sigurna bude i optimizovana (korišćenje asimetricne enkripcije je dosta sporije, dok se kod simetricne enkripcije nekako mora poslati kljuc). Time se smanjuje rizik od presretanja, neovlašćenog pristupa i modifikacije informacija tokom prenosa.

2.4 Rate limiting kao deo zaštite

Da ne bi doslo do prekomernog "bombardovanjašervera i kako bi se očuvala dostupnost sistema potrebno je implementirati mehanizme za ograničavanje broja zahteva koje korisnici mogu da pošalju u nekom određenom vremenskom preiodu.

Ovo nam pomaže u prevenciji zloupotreba poput DoS napada ili prekomernog korišćenja resursa servera, čime bi se mogla izazvazi velika degradacija performasni samog servera.

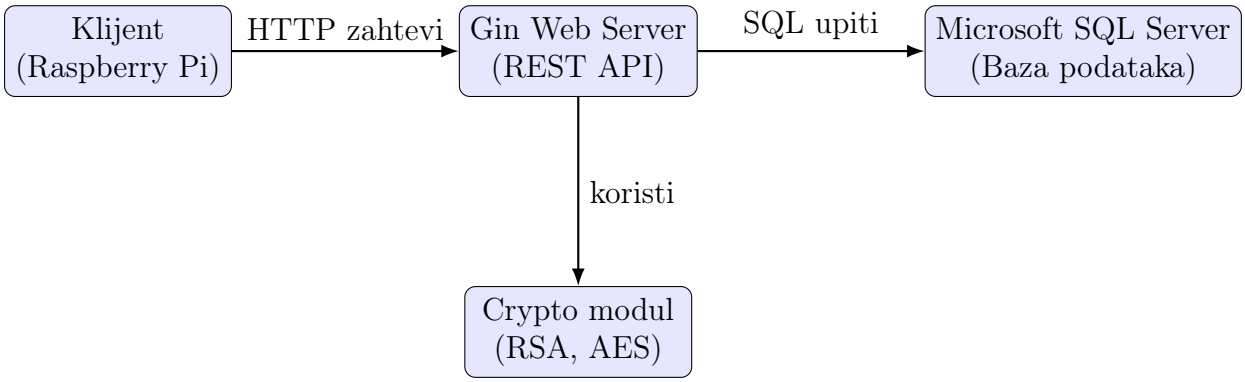
3. Arhitektura sistema

3.1 Pregled komponenti

Projekat *GoGuard* organizovan je u jasno definisane pakete koji odražavaju osnovne funkcionalne celine sistema. Struktura repozitorijuma olakšava održavanje i dalji razvoj, dok istovremeno omogućava modularnost i jasnu separaciju odgovornosti:

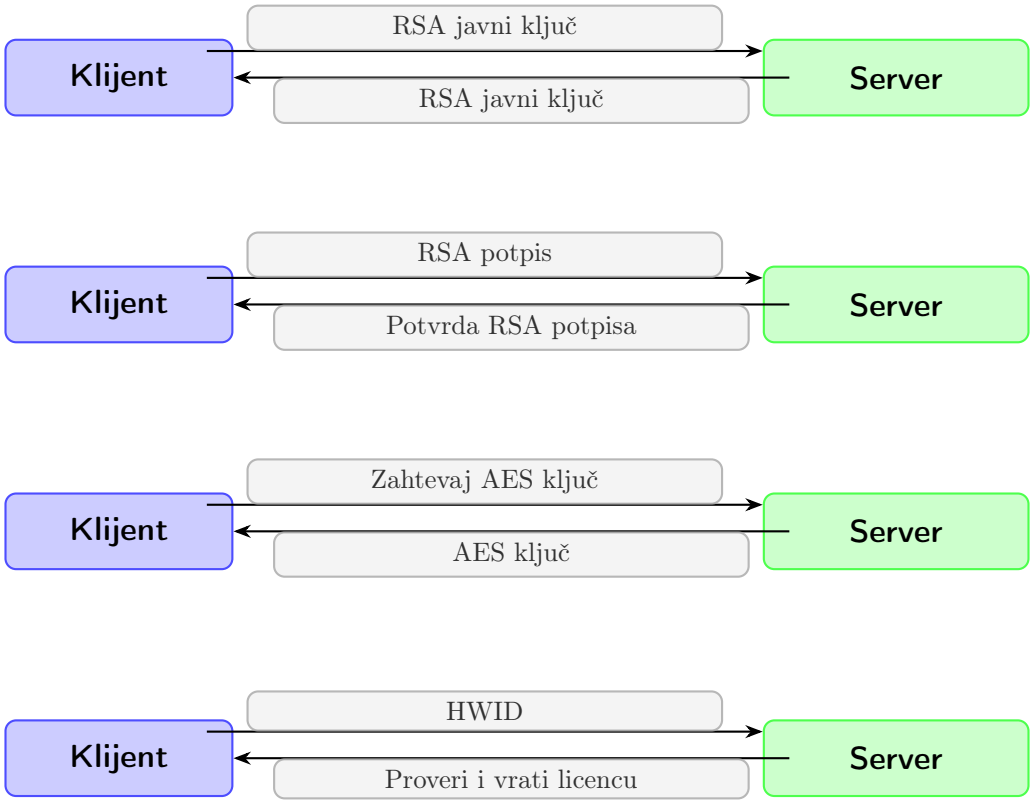
- **crypto/** - implementacije kriptografskih operacija, uključujući RSA i AES algoritme za enkripciju i dekripciju podataka. Tu se nalaze i funkcije za upravljanje licencama sa kriptografskim potpisima, kao i odgovarajući testovi koji osiguravaju ispravnost implementacije.
- **db/** – Paket zadužen za komunikaciju sa bazom podataka. Implementira SQL upite i metode za pristup podacima o korisnicima, licencama i hardverskim identifikatorima. U ovom projektu je korišćen Microsoft SQL Server.
- **models/** – Definisane su strukture podataka koje se koriste za prenos i primanje poruka kroz REST API. Ove strukture predstavljaju JSON formate za handshake poruke, digitalne potpise, zahteve za licence, kao i modele podataka za uređaje sa pripadajućim poljima za identifikaciju i licence.
- **server/** – Sadrži serverski kod i rutere za API. Implementirani su handleri za različite REST zahteve, učitavanje konfiguracije iz config.json, mehanizmi za rate limiting, handshake protokol za sigurno uspostavljanje sesije, kao i upravljanje AES ključevima i digitalnim potpisima.
- **config.json** – Konfiguracioni fajl u JSON formatu koji sadrži parametre za povezivanje na bazu podataka (server, korisničko ime, lozinku, ime baze i tabelu), adresu na kojoj API server radi, kao i serverski ključ korišćen za generisanje i proveravanje licence.
- **main.go** – Ulazna tačka aplikacije. Učitava konfiguraciju, uspostavlja konekciju sa bazom podataka, kreira instancu serverske aplikacije, registruje rute i pokreće Gin web server na zadatoj adresi. Takođe podešava logovanje u fajl za lakšu dijagnostiku i praćenje rada sistema.

3.2 Dijagram sistema



Slika 3.1: Dijagram osnovnih komponenti sistema GoGuard

3.3 Tok komunikacije



Slika 3.2: Tok komunikacije između klijenta i servera u GoGuard sistemu

Treba uzeti u obzir da klijent razmenu mora inicirati prvim zahtevom u kom će slati RSA Javni Ključ, prilikom kog će kao odgovor uz javni ključ servera dobiti i neki **SessionID** koji će na dalje koristiti u okviru svakog requesta da bi server mogao da prepozna sa kim ima kontakt.

4. Jezičke specifičnosti Go-a

4.1 Gin framework

Gin je HTTP web framework napisan u programskom jeziku Go (Golang). Ima API sličan Martini framework-u, ali sa performansama koje su do 40 puta bolje od Martini-ja, i kreatori sami kažu "Ako ti treba vrhunska performansa, koristi Gin (*If you need smashing performance, get yourself some Gin*)". Korišćenje **httprouter**-a je važan razlog za ovo. Koristeći zvaničnu dokumentaciju možemo videti primer :

```
func main() {
    r := gin.Default()
    r.GET("/ping", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{
            "message": "pong",
        })
    })
    r.Run() // listen and serve on 0.0.0.0:8080
}
```

Ovaj primer je dosta jednostavan, može se podeliti u 3 koraka:

1. Korišćenje `gin.Default()` za pravljenje `Engine` objekta sa default konekcijama
2. Registrovanje funkcije za adresu "/ping"u okviru `GET` metode `Engine`-a
3. Pokretanje `Engine`-a da sluša na port i odrađuje posao

Pored standardnih, najčešće korišćenih HTTP metoda (`GET`, `POST`, `PUT`, `DELETE`) treba napomenuti da Gin takođe obezbeđuje interface `ANY` koji može direktno da poveže sve metode obrade HTTP zahteva na jednu adresu.
Kao što možemo videti u primeru iznad, `gin.Default()` koristi se prilikom pravljenja `Engine`-a, koji je samo wrapper funkcija za `New`, što možemo primetiti jer se `Engine` zapravo pravi korišćenjem `New` interface-a.
Pored osnovnog rada sa rutama, Gin omogućava korišćenje middleware funkcija — delova koda koji se izvršavaju pre ili posle obrade zahteva. Middleware se često koristi za logovanje, autentikaciju, validaciju i merenje performansi.

- **Logger** – loguje svaki zahtev koji dolazi na server.
- **Recovery** – sprečava pad servera u slučaju panic-a i vraća HTTP 500 odgovor.

Middleware se može dodati ručno pomoću funkcije Use(), na sledeći način:

```
r := gin.New()
r.Use(gin.Logger())       // Dodavanje Logger middleware-a
r.Use(gin.Recovery())     // Dodavanje Recovery middleware-a
```

U projektu je dodat prilagođeni middleware za ograničavanje broja zahteva po IP adresi (Rate Limiting), implementiran korišćenjem paketa `golang.org/x/time/rate`. Middleware je registrovan u ruteru pomoću:

```
s.router.Use(RateLimiterMiddleware())
```

Na ovaj način svaka IP adresa ima definisan maksimalan broj dozvoljenih zahteva u jedinici vremena, čime se sprečava preopterećenje sistema.

Pored jednostavnih ruta kao u gornjem primeru, Gin omogućava i rad sa parametarskim rutama, koje se definišu pomoću : za jedan parametar ili * za hvatanje ostatka putanje.

```
r.GET("/user/:id", func(c *gin.Context) {
    id := c.Param("id")
    c.JSON(http.StatusOK, gin.H{"user_id": id})
})
```

Adresa i port na kojima server sluša mogu se precizirati prosleđivanjem stringa u `Run()` metodu:

```
r.Run("192.168.1.100:9090") // sluša samo na navedenoj adresi i portu
```

Takođe handler funkcija u Gin-u je funkcija sa potpisom `func(c *gin.Context)`, gde je c pokazivač na gin.Context. Ovaj objekat daje pristup podacima o zahtevu i metodama za slanje odgovora.

Ukoliko ne želimo eksplicitno da koristimo baš funkciju ovakvog potpisa uvek možemo u okviru nete rute dodeliti i funkciju bez parametara koja će samo vraćati `func(c *gin.Context)`

```
s.router.POST("/handshake", s.HandleHandshake())
.
.
.
func (s *Server) HandleHandshake() gin.HandlerFunc {
    return func(context *gin.Context) { ... }
}
```

4.2 Concurrency model u Gin-u

Gin framework, kao i standardna Go biblioteka net/http, obrađuje svaki HTTP zahtev u posebnoj **gorutini**. To omogućava istovremen rad sa više zahteva i bolje iskorišćenje multi-core arhitekture. Kako navodi blog "*Guide To Building Fast Backends In Gin*", "*By spawning a new goroutine for each incoming request, Gin can serve many clients concurrently*".
Goroutine se tako nazivaju jer postojeći termini — niti, korutine, procesi i slični — nose

netačne konotacije.
Gorutina ima jednostavan model: to je funkcija koja se izvršava konkurentno sa drugim gorutinama u istom adresnom prostoru, takođe su veoma "lake" memorijski.
Gorutine se multipleksiraju na više OS niti, tako da, ako neka od njih blokira (npr. prilikom čekanja na I/O operaciju), ostale nastavljaju sa radom.
Njihov dizajn sakriva većinu složenosti koja je inače prisutna kod kreiranja i upravljanja nitima.
Da bi se funkcija ili metoda pokrenula kao gorutina, ispred njenog poziva dodaje se ključna reč **go**. Nakon što se izvršavanje završi, gorutina tiho izlazi iz rada. Efekat je sličan Unix shell operatoru &, koji komandu pokreće u pozadini.

```
go list.Sort()  // pokreće list.Sort konkurentno, bez čekanja na završetak
```

4.3 Mutexi

Ukoliko više gorutina pristupa određenom deljenom resursu u cilju čitanja ili pisanja tu možemo imati problem jer može doći do nedefinisanog ponašanja/rezultata.
Baš iz tog razloga potrebno je koristiti neki vid žaključavanja"tog resursa koji bi u slučaju našeg projekta bi to bilo neko "polješessions koji je mapa koja preslikava sessionId u pokazivač na Sessiju.

```
type Server struct {
 router    *gin.Engine
 db        *sql.DB
 sessions  map[string]*Session <-----------------------
 mutex     sync.RWMutex <-----------------------------
 ServerKey string
}
```

Baš kada njoj pristupamo u nekoj Handler funkciji ćemo koristiti sync.RWMutex.
`sync.RWMutex` je tip iz Go standardne biblioteke koji omogućava sinhronizaciju pristupa deljenim resursima u konkurentnim programima.
RW (Read-Write) mehanizam dozvoljava da:

- više gorutina istovremeno obavlja operacije čitanja (*read lock*)
- operacije pisanja (*write lock*) može vršiti isključivo jedna gorutina i to samo kada nema aktivnih čitača

Ovaj pristup je posebno efikasan u scenarijima gde čitanje podataka značajno nadmašuje pisanje, jer omogućava veću paralelnost bez narušavanja integriteta podataka.
Primer zaključavanja za čitanje iz projekta:

```
func (s *Server) HandleAESKey() gin.HandlerFunc {
 return func(context *gin.Context) {
  .
        .
        .
  s.mutex.RLock() // Zaključavanje za čitanje
```

```
    session, sessionExists := s.sessions[sessionIdString]
    var expiredSession bool
    if sessionExists {
     expiredSession = s.sessions[sessionIdString].IsExpired()
    }
    s.mutex.RUnlock()
          .
          .
          .
}
```

4.4 Paket crypto

Paket crypto u programskom jeziku Go predstavlja skup standardnih biblioteka koje pružaju osnovne kriptografske funkcionalnosti (`https://pkg.go.dev/crypto`). U okviru ovog paketa nalaze se implementacije različitih kriptografskih algoritama, kao što su **simetrična enkripcija** (npr. AES), **asimetrična enkripcija** (npr. RSA), **heš funkcije** (npr. SHA-256), digitalni potpisi i generisanje sigurnih nasumičnih brojeva. Paket je dizajniran da bude siguran i efikasan, i koristi se za zaštitu podataka, autentifikaciju i proveru integriteta u različitim aplikacijama. Programeri ga često koriste za implementaciju bezbednosnih protokola, enkripciju komunikacije i čuvanje poverljivih informacija.

4.5 SQL Server integracija

U Go aplikacijama, integracija sa Microsoft SQL Server bazom podataka se najčešće ostvaruje korišćenjem drajvera go-mssqldb. Ovaj drajver omogućava komunikaciju sa SQL Server-om preko standardnog database/sql interfejsa u Go jeziku.
Proces integracije obično uključuje:

- Uvoz go-mssqldb drajvera u projekat.

```
import (
 "database/sql"
 _ "github.com/denisenkom/go-mssqldb"
}
```

- Otvaranje konekcije ka bazi pomoću funkcije sql.Open, gde se navodi mssql kao drajver.

```
connString := "server=localhost;user id=sauser;
            password=lozinka;database=moja_baza"
db, err := sql.Open("mssql", connString)
```

- Izvršavanje SQL upita i komandi korišćenjem funkcija iz paketa database/sql.
- Upravljanje konekcijama i rukovanje eventualnim greškama.

Ovakav pristup omogućava efikasno i standardizovano korišćenje SQL Server baze u Go aplikacijama, uz korišćenje poznatog i dobro podržanog database/sql paketa.

4.6 Error handling

U Go programskom jeziku, greške se ne tretiraju kao izuzeci (exceptions), već kao vrednosti koje funkcije vraćaju zajedno sa rezultatima. Najčešće, funkcije vraćaju dva ili više rezultata, gde je poslednji rezultat tipa error. Provera i obrada greške vrši se odmah nakon poziva funkcije.
Primer iz projekta:

```
ciphertext, err := base64.StdEncoding.DecodeString(msg)
if err != nil {
    fmt.Println("decoding string failed")
    return "", err
}
```

5. Objašnjenje dizajnerskih odluka

5.1 Izbor programskog jezika Go

Golang je izabran kao glavni programski jezik zbog svoje jednostavnosti, efikasnosti i podrške za konkuretno programiranje.Baš iz razloga što poseduje sposobnost da lako pravi konkurentne aplikacije omogućava nam da imamo skalabilne i preformansno optimizovane servere.
Još jedan razlog jeste bogata standardna biblioteka. Tokom razvoja projekta, sve potrebene biblioteke i paketi bili su vrlo lako dostupni i odlični za korišćenje uz dobro napisanu dokumentaciju, u to podrazumevamo podršku za rad sa HTTP protokolom, kriptografijom, sinhronizacijom...

5.2 Izbor Gin frameworka

- **Jednostavnost korišćenja**: Gin ima jednostavan i intuitivan API koji olakšava brzo pravljenje REST API-ja i web servera
- **Visoke performanse**: Zbog korišćenja httprouter-a i efikasnog rukovanja gorutinama, Gin postiže odlične performanse
- **Validacija zahteva i binding**: Gin ima ugrađenu podršku za validaciju podataka i automatsko mapiranje JSON, XML ili form podataka na Go strukture.

```
type DigitalSignatureMessage struct {
 Payload   string `json:"Payload"`
 Signature string `json:"Signature"`
}
```

  npr. dovoljno je samo prilikom pravljenja strukture navesti kako da se ona mapira u JSON

- **Middleware podrška**: Gin omogućava lako dodavanje middleware funkcija za logovanje, autentikaciju, obradu grešaka i druge

5.3 Izbor kriptografskih algoritama

Prilikom odabira kriptografskih algoritama korišćena su dva pristupa: asimetrični algoritam **RSA** i simetrični algoritam **AES-256-GCM**.

RSA, kao asimetrični algoritam, koristi par ključeva — javni i privatni. Javni ključ može biti slobodno distribuiran i koristi se za enkripciju podataka, dok se privatni ključ koristi za njihovu dekripciju. U okviru sistema, klijent šalje svoj RSA javni ključ, koji server koristi za verifikaciju identiteta klijenta putem digitalnog potpisa.

Nakon uspešne verifikacije, server generiše simetrični ključ AES-256-GCM, koji je efikasniji za obradu većih količina podataka. Pošto se kod simetrične enkripcije koristi isti ključ i za enkripciju i za dekripciju, AES ključ ne može biti poslat u otvorenom obliku.

Zato se AES ključ enkriptuje RSA javnim ključem klijenta i tako šalje nazad, što omogućava da samo klijent, koji poseduje odgovarajući privatni ključ, može da ga dekriptuje i koristi. Nakon toga, komunikacija se nastavlja koristeći AES-256-GCM za sigurnu i efikasnu razmenu podataka.

Osobina	**RSA (Asimetrična)**	**AES-256-GCM (Simetrična)**
Tip ključeva	Javni i privatni	Jedan zajednički ključ
Brzina	Sporija	Brža
Kompleksnost	Veća računarska zahtevnost	Manja računarska zahtevnost

Tabela 5.1: Tabela osnovnih osobina RSA i AES-256-GCM enkripcija

6. Zaključak

6.1 Rezime funkcionalnosti sistema

Sistem omogućava sigurnu komunikaciju između klijenta i servera korišćenjem kombinacije asimetrične i simetrične enkripcije. Prvo se razmenjuju RSA javni ključevi radi verifikacije i uspostavljanja poverenja, nakon čega se preko sigurne RSA veze šalje AES-256-GCM ključ za brzu i efikasnu enkripciju podataka. Svi osetljivi podaci, uključujući hardverski ID (HWID) i informacije o licenci, razmenjuju se preko AES-kriptovane veze, čime je komunikacija dodatno zaštićena.

Sistem koristi Gin web framework za efikasno upravljanje HTTP zahtevima i podržava middleware za dodatne bezbednosne mehanizme kao što su ograničenje broja zahteva po IP adresi. Kroz implementaciju konkurentnosti u Go jeziku, sistem može istovremeno obrađivati veliki broj zahteva, obezbeđujući skalabilnost i stabilnost.

Baza podataka čuva isključivo podatke vezane za licence. Nakon primanja šifrovanog HWID-a od klijenta, sistem proverava da li licenca za taj HWID već postoji. Ukoliko licenca ne postoji, sistem je generiše, šifruje i čuva u bazi, a zatim vraća klijentu. Ova logika omogućava efikasno upravljanje licencama i osigurava validnost pristupa klijenata.

Za zaštitu kritičnih sekcija u kodu koristi se `mutex` mehanizam, čime se sprečavaju konflikti prilikom istovremenog pristupa deljenim resursima.

6.2 Resursi

https://www.ninjaone.com/blog/how-to-check-hwid-on-your-device/
https://dev.to/leapcell/a-deep-dive-into-gin-golangs-leading-framework-5e39
https://gin-gonic.com/en/docs/
www.jetbrains.com/guide/go/tutorials/rest_api_series/gin/
https://slashdev.io/blog/guide-to-building-fast-backends-in-gin-golang-in-2024-2
https://go.dev/doc/effective_go#goroutines