SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 3058

Implementacija PSD2 poslužitelja

Antonio Špoljar

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 3058

Pristupnik:	**Antonio Špoljar (0036509708)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	doc. dr. sc. Stjepan Groš

Zadatak: **Implementacija PSD2 poslužitelja**

Opis zadatka:

PSD2 je direktiva Europske unije koja je stupila na snagu u rujnu 2019. godine. Navedenom direktivom Europska unija traži od financijskih institucija, prvenstveno banaka, da omoguće pristup svojim poslovnim sustavima putem API-ja, kako bi se omogućilo nuđenje financijskih usluga svim tvrtkama, te na taj način liberaliziralo financijsko tržište. Međutim, otvaranje novog komunikacijskog kanala koji zadire u samu jezgru IT sustava financijskih institucija otvara novi potencijalni vektor napada o kojemu treba voditi računa. U sklopu diplomskog rada potrebno je razviti poslužiteljsku komponentu PSD2 sustava koja prihvaća REST API zahtjeve klijenata, te ih potom šalje u RabbitMQ red poruka. Poslužiteljska komponenta treba voditi računa o ispravnosti upita, te o zahtjevima koji su poslani u red poruka, ali na koje nije odgovoreno. Također, poslužiteljska komponenta treba ima mogućnost praćenja stanja sustava, te odgovarajući GUI putem kojega je moguće pratiti to stanje. Radu priložiti izvorni kod razvijenih i korištenih programa. Citirati korištenu literaturu i navesti dobivenu pomoć.

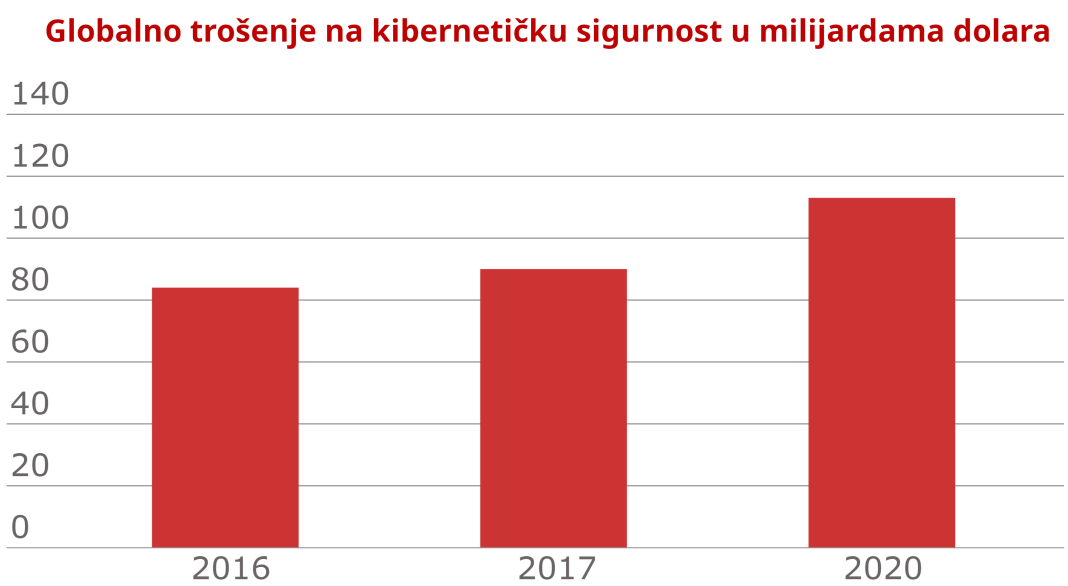
Rok za predaju rada: 27. lipnja 2022.

SADRŽAJ

Zahvaljujem se doc. dr. sc. Grošu na ukazanom povjerenju i pomoći pri izradi rada.

1. Uvod

Sigurnost igra sve veću ulogu u informatičkom svijetu. U financijskom sektoru se zbog rukovanja osjetljivim podacima korisnika pridjeljuje posebna važnost na kibernetičku sigurnost (engl. *cybersecurity*). Svake godine banke troše sve više na kibernetičku sigurnost zbog pritiska korisnika i nadležnih organizacija [8], ali i zbog velikih šteta ukoliko se dogodi uspješan kibernetički napad. Slika 1.1 prikazuje porast izdvajanja banaka za kibernetičku sigurnost zadnjih godina. Jedna od direktiva nadležnih tijela Europske unije koja podupire sigurnost je revidirana direktiva o platnim uslugama (engl. *Payment Service Directive 2*, skraćenica PSD2) [7]. Uvedena 2015. godine, PSD2 zamjenjuje i nadograđuje prijašnju PSD direktivu iz 2007. godine s ciljem poboljšanja sigurnosti transakcija, zaštite korisnika te povećanja kompetitivnosti tržišta. Najveća promjena što se tiče banaka je otvaranje svojih usluga za plaćanje drugim kompanijama, trećim pružateljima platnih usluga (engl. *Third Party Payment Services Providers*, skraćenica TPP).



Slika 1.1: Globalno trošenje banaka na kibernetičku sigurnost [19]

Uvođenjem PSD2 direktive došlo je do potrebe za prilagodbom postojećih aplikacijskih programskih sučelja (engl. *Application user interface*, skraćenica API) banaka. Izrađeno je nekoliko standarda, a u hrvatskim je bankama usvojen standard Berlinske grupe pod imenom NextGenPSD2 [3]. Posebna radna skupina NextGenPSD2 unutar Berlinske grupe razvila je moderan, otvoren i efikasan istoimeni standard s naglaskom na sigurno dohvaćanje podataka [9]. NextGenPSD2 pruža sučelje za pristup računu (engl. *Access to Account*, skraćenica XS2A). XS2A je sučelje koje se nalazi ispred drugih API-ja i služi za interakciju s procesima, podatcima i infrastrukturama definiranim u PSD2 direktivi.

Otvaranjem svojih usluga za plaćanje, banke se izlažu potencijalnim napadima. Moguće rješenje za to je postavljanje posredničke sigurnosne stijene (engl. *proxy firewall*) ispred samog API-ja banke. Na taj način se osigurava integritet i sprječavaju potencijalni napadi. Sigurnosna stijena za PSD2 opisana u ovom radu se sastoji od 3 glavna dijela: poslužitelja, koji prihvaća HTTP zahtjeve TPP-a, dijela za filtriranje, koji odvaja maliciozne zahtjeve, te klijenta, koji prosljeđuje zahtjeve banci. Tri komponente međusobno komuniciraju putem RabbitMQ asinkronih redova. Sličan je i povratni proces, gdje klijent prihvaća odgovor banke, prosljeđuje ga dijelu za filtriranje, koji ga šalje poslužitelju. Naposlijetku, poslužitelj vraća odgovor pošiljatelju, odnosno TPP-u.

Kako bi se testirala ispravnost, ali i stupanj zaštite od malicioznih radnji, potrebno je testirati poslužitelj. To se napravilo pomoću skripte koja šalje zahtjeve prema unaprijed definiranom toku (engl. *flow*), ali dopušta modifikaciju svakog pojedinog zahtjeva. Na taj način je moguće testirati ponašanje poslužitelja u slučaju slanja neregularnih zahtjeva.

Cilj ovog rada je izrada poslužitelja prethodno opisane sigurnosne stijene. Poslužitelj PSD2 sigurnosne stijene ima zadaću primanja zahtjeva TPP-a i slanja odgovora na te zahtjeve. Pošto se zahtjev prosljeđuje na daljnju obradu u ostatak sigurnosne stijene i onda se čeka na odgovor, taj zahtjev je potrebno zapamtiti u bazi podataka. Moraju se moći prihvatiti svi zahtjevi koje bi inače primao API banke, a primljene zahtjeve je potrebno proslijediti u RabbitMQ red, gdje ga može dohvatiti dio za filtriranje. Proces prihvaćanja zahtjeva i vraćanja odgovora se nastavlja kao što je prethodno opisano, a poslužitelj mora čekati odgovor banke koji mu kroz drugi RabbitMQ red šalje dio za filtriranje. Zatim se iz poslužitelja odgovor vraća TPP-u. Pritom je potrebno posebnu pažnju pridijeliti sigurnosnim karakteristikama cijelog procesa.

Rad je podijeljen na 7 glavnih poglavlja. U drugom poglavlju se ukratko opisuje PSD2 direktiva, NextGenPSD2 standard i objašnjavaju se svi tehnički pojmovi vezani uz sudionike procesa definiranih u PSD2. Sljedeće poglavlje opisuje sigurnosnu stijenu, njenu arhitekturu, kao i zadaće i funkcionalnosti njenih komponenata, odnosno RabbitMQ redova i klijenta. Četvrto poglavlje detaljno opisuje poslužitelj, posebice način na koji prihvaća zahtjeve i šalje odgovore. Peto poglavlje se bavi testiranjem poslužitelja putem specijalizirane Python skripte.

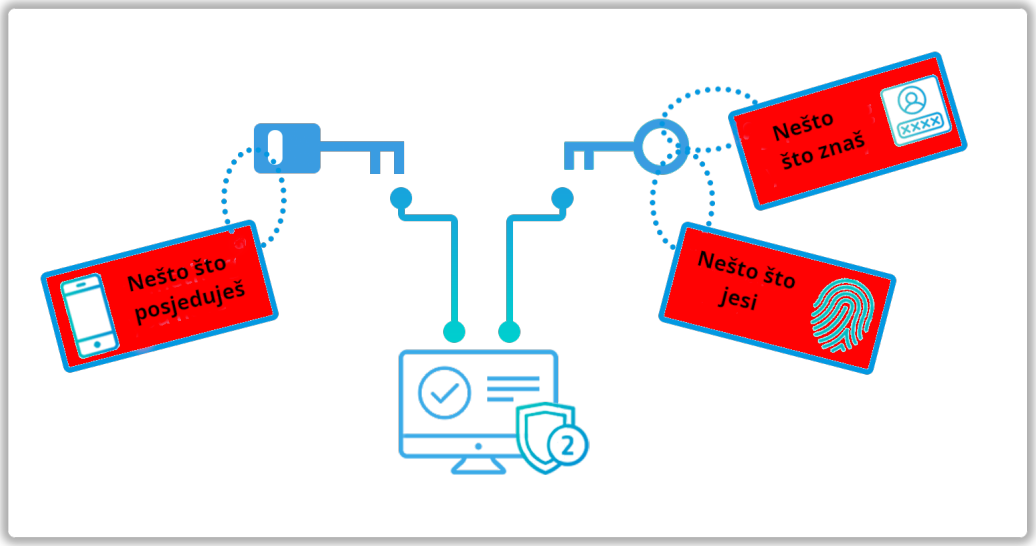
2. PSD2

U ovom poglavlju se opisuje PSD2 direktiva i njene specifičnosti. Objašnjeno je što je to NextGenPSD2 i koje nove usluge uvodi. Predstavljeni su svi bitniji sudionici procesa opisanih PSD2 direktivom.

2.1. PSD2 direktiva

PSD2 direktiva nastala je 2015. godine iz potrebe za promjenom tada aktualne PSD direktive. Krajnji ciljevi su integriranije i učinkovitije tržište plaćanja, povećanje tržišnog natjecanja i transparentnosti, poboljšanje sigurnosti transakcija, zaštita korisnika platnih usluga (engl *Payment Service User*, skraćenica PSU) i lakše rukovanje s više otvorenih računa koji glase na istog PSU-a. Najbitnija posljedica za banke je otvaranje svojih usluga za plaćanje drugim kompanijama, poput Revoluta i Google Paya.

Jedna od metoda dodatne zaštite transakcija je pouzdana autentifikacija klijenta (engl. *Strong Customer Authentication*, SCA). Uvođenje SCA znači da se za pristupe povjerljivim podacima, kao što su podaci o računu ili izvršavanje isplate, treba autentificirati pomoću dvofaktorske autentifikacije (engl. *Two Factor Authentication*, skraćenica 2FA). 2FA uvodi još jedan korak pri autentifikaciji, što višestruko otežava krađu povjerljivih podataka. Kao što je prikazano na slici 2.1, drugi korak je obično definiran nečime što korisnik zna (npr. sigurnosno pitanje), nečime što posjeduje (npr. mobilni uređaj) ili nečime što jest (npr. otisak) [4].



Slika 2.1: Pitanja u dvofaktorskoj autentifikaciji

PSD2 sadrži i pravila o novim uslugama koje koriste treći pružatelji platnih usluga u ime korisnika platnih usluga. Prema [12], nove usluge su sljedeće:

- usluga iniciranja plaćanja (engl. *Payment Initiation Service*, skraćenica PIS) koju mora podržavati TPP pružatelj usluge iniciranja plaćanja (Payment Initiation Service Provider, PISP)
- usluga informiranja o računu (engl. *Account Information Service*, AIS) koju mora podržavati pružatelj usluga informiranja o računu koji svojem klijentu nudi AIS (engl. *Account Information Service Provider*, skraćenica AISP)
- usluga potvrde raspoloživosti sredstava (engl. *Confirmation on the Availability of Funds Service*, skraćenica FCS) koju mora podržavati PISP

PISP dozvoljava da TPP, nezavisna kompanija, radi uplate i isplate u ime PSU-a. AISP pak pristupa podacima o računu, komunicira s bankom i prikazuje informacije o računu, sve u ime korisnika platnih usluga. Da bi pružili ove usluge, banke, odnosno pružatelji platnih usluga koji vode račun (*Account Servicing Payment Service Provider*, skraćenica ASPSP) moraju mijenjati svoje postojeće API-je [11]. PPISP je na primjer Revolut [17], a primjer AISP-a su aplikacije za pomoć oko organiziranja financija, prikaza uplata i isplata i sl.

Za provođenje usluga uz pristanak PSU-a, TPP mora pristupati računu PSU-a. Pružatelj platnih usluga (*Payment Service Provider*, skraćenica PSP), odnosno pružatelj platnih usluga koji vodi račun (ASPSP, najčešće banke), je zadužen za vođenje računa. Kako bi se olakšao pristup i pregled računa, svaki ASPSP mora imati prikladno sučelje za pristup računu XS2A. Rad sučelja, kao i odgovornosti i prava TPP-a i ASPSP-a, definiran je direktivom PSD2 [12].

Osim PSD2, još su dva dokumenta koja čine osnovu za regulatorne zahtjeve. Prvi je *Regulatorni tehnički standardi* (skraćenica RTS) koji definira na koji način treba vršiti pouzdanu autentifikaciju klijenta te kako koristiti zajedničke i sigurne otvorene standarde komunikacije (engl. *Common And Secure Communication*, skraćenica CSC). Drugi je *Zakon o platnom prometu* (skraćenica ZPP), lokalni zakon kojim se direktiva prenosi u zakonodavstvo [12].

2.2. NextGenPSD2

NextGenPSD2 je standard nastao inicijativom istoimene posebne radne grupe u okviru Berlinske grupe. Cilj je stvaranje jedinstvenog, otvorenog i usklađenog europskog standarda za aplikacijsko programsko sučelje, putem kojeg bi treći pružatelji usluga pristupali računima, transakcijama i ostalim podacima u skladu s PSD2. Ključni dio je XS2A sučelje koje služi za interakciju TPP-a sa cjelokupnim API-jem. NextGenPSD2 pridonosi napretku prema jedinstvenom europskom platnom tržištu, od kojeg će posebne koristi imati potrošači i poduzeća.

Prema [12], neke od ključnih karakteristika NextGenPSD2 su:

- REST API koji koristi HTTP/1.1 uz TLS protokol 1.2 ili viši
- identifikacija TPP-a putem certifikata: QWAC (engl. *Qualified website authentication certificate*) je osnovna mjera zaštite od raspodijeljenog napada uskraćivanja usluge (engl *distributed denial of service*, skraćenica DDOS) te je obavezan, a QSEAL (engl. *Qualified Seal Certificates*) je izboran za banke, dok su TPP-i dužni pratiti upute pojedine banke
- podržava sve slučajeve pružanja usluga informiranja o računu, iniciranja plaćanja, usluga potvrde raspoloživosti sredstava, dok je podržavanje budućih, višestrukih i periodičnih plaćanja izborno

- podrška viševalutnih računa
- četiri vrste pouzdane autentifikacije klijenta: preusmjeravanje, OAuth2, odvojeno i ugrađeno
- podrška za račune koji služe za namirenje kartičnih transakcija
- transparentna struktura koja čak i u složenim poslovnim procesima mora omogućiti jednostavan pregled i dohvat resursa
- podatkovne strukture u obliku JSON-a ili XML-a
- mogućnost za dodatna proširenja koja uvode nove usluge

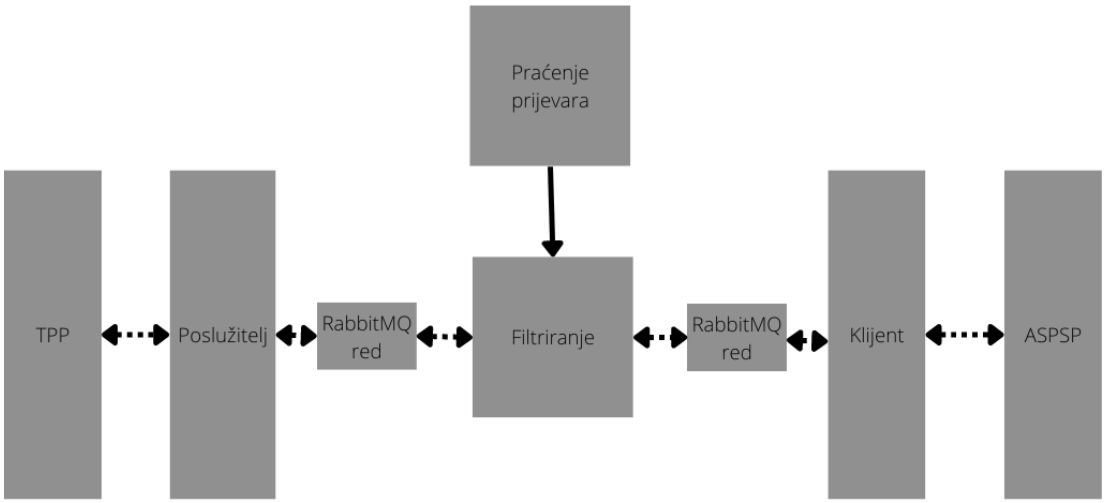
3. Sigurnosna stijena

U ovom poglavlju se opisuje posrednička sigurnosna stijena te se pojašnjavaju veze između komponenti koje ju grade. Najprije je objašnjena arhitektura i funkcionalnost svakog pojedinog dijela, a potom su pobliže opisane dvije od tri najbitnije komponente. Potpoglavlje o RabbitMQ asinkronim redovima najprije prikazuje pokretanje redova putem platforme Docker, a zatim gradi jednostavan demonstracijski program pomoću kojeg se prezentira rad i upotreba redova. Na kraju se prikazuje kako su se redovi koristili u sigurnosnoj stijeni. U zadnjem potpoglavlju se prolazi kroz funkcionalnosti klijenta sigurnosne stijene, bez ulaženja u sve detalje izvedbe.

3.1. Arhitektura

Sigurnosna stijena za NextGenPSD2 sučelja služi za zaštitu sučelja ASPSP-a od potencijalnih napada i pogrešnih ili malicioznih zahtjeva. Kôd je dostupan na repozitoriju [10]. Arhitektura sigurnosne stijene opisane u ovom radu je prikazana na slici 3.1. Sustav je zamišljen kao prilagodljiva kompozicija mikroservisa gdje su komponente neovisne jedna o drugoj i komuniciraju pomoću definiranih sučelja. To omogućuje buduću zamjenu ili preinaku pojedinih komponenti bez utjecanja na cjelokupni sustav. Također, takva podjela omogućava bolje prilagodbe po pitanju skalabilnosti.

Jedan je glavni proces obrade zahtjeva koji su upućeni sigurnosnoj stijeni, no on teče u dva smjera: prvi je od TPP-a do ASPSP-a, a drugi suprotan. Uobičajeno je da se oba smjera procesa izvršavaju zaredom, osim u slučaju nevaljanog zahtjeva, kada sigurnosna stijena ne propušta zahtjev do sučelja ASPSP-a. Tipičan primjer komunikacije je provođenje isplate. TPP zatraži u ime PSU-a isplatu s računa, što je prvi spomenuti smjer procesa. Nakon obrade, ASPSP vraća odgovor TPP-u u drugom smjeru. Ovakav i slični procesi moraju teći neometano kroz sigurnosnu stijenu bez da TPP bude svjestan prisutnosti dodatne komponente u procesu komunikacije s ASPSP-om.



Slika 3.1: Arhitektura sigurnosne stijene

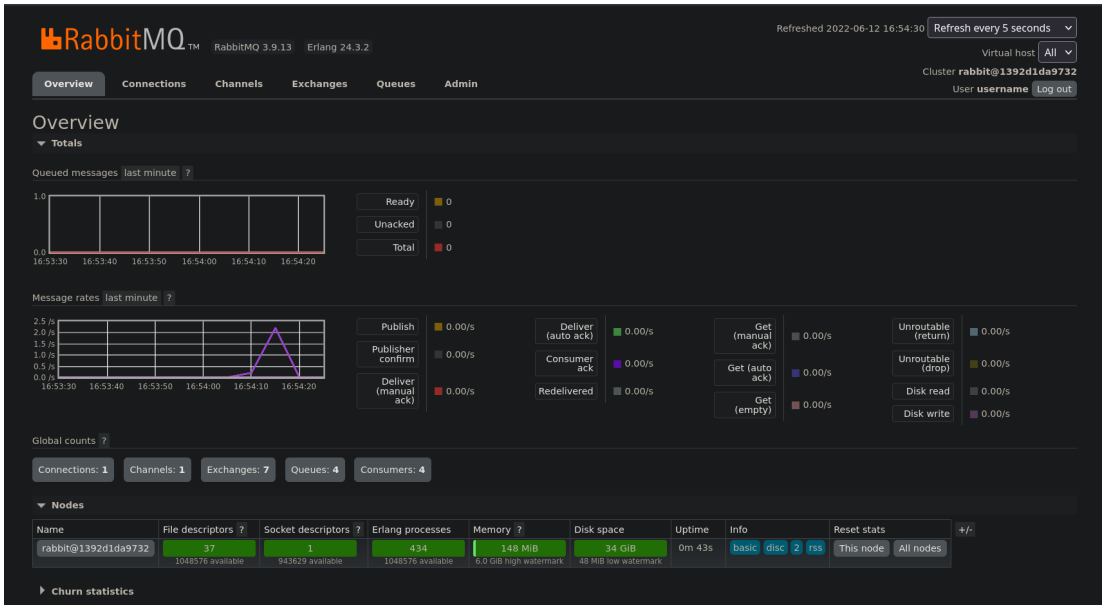
Pružatelj trećih usluga (TPP) je uobičajeno kompanija koja ima sučelje preko kojeg se izvršavaju plaćanja, pregled računa i slične radnje. TPP šalje HTTP zahtjeve kakve bi inače slao ASPSP-u. Zahtjevi prema PSD2 direktivi moraju biti zaštićeni protokolom TLS (engl. *Transport Layer Security*) 1.2 ili viši. TLS je kriptografski protokol dizajniran kako bi očuvao sigurnost u komunikacijskim kanalima. Aktualna verzija je 1.3, no veliki broj sustava i dalje koristi verziju 1.2, zbog čega u sigurnosnoj stijeni obje verzije moraju biti podržane.

Poslužitelj mora imati sučelje identično ili vrlo slično API-ju ASPSP-a, kako bi TPP-i mogli komunicirati s njime kao i s pravim ASPSP-om. Nakon što TPP pošalje zahtjev, u poslužitelju se isti parsira i obradi, te se šalje u prvi RabbitMQ red, nazvan *FILTER_REQUEST_QUEUE*. Dio za filtriranje prihvaća poruku iz reda, obrađuje ju i uklanja pogrešne i potencijalno maliciozne zahtjeve. Pri tome se koristi komponentom za praćenje i detekciju prijevara. Ako zahtjev nije označen kao pogrešan ili maliciozan, šalje se pomoću poruke u drugi RabbitMQ asinkroni red, nazvan *BANK_REQUEST_QUEUE*. Iz tog reda čita klijent, koji pomoću dijelova pročitane poruke sastavlja zahtjev. Taj zahtjev je identičan početnom HTTP zahtjevu koji je slao TPP u svim bitnim zaglavljima i parametrima, kao i u podacima koji se šalju u tijelu. Time završava prvi smjer procesa.

Drugi smjer počinje primitkom odgovora ASPSP-a u klijentu sigurnosne stijene. Klijent radi sličan postupak kakav je u prvom smjeru radio poslužitelj, odnosno parsira odgovor i komponente sprema u poruku, koju zatim šalje trećem RabbitMQ redu, nazvanom *BANK_RESPONSE_QUEUE*. Komponenta za filtriranje prebacuje poruku iz trećeg reda poruka u četvrti, nazvan *FILTER_RESPONSE_QUEUE*, pri čemu je ne mijenja, no može zabilježiti i pohraniti potrebne podatke. Poslužitelj sluša četvrti red (*FILTER_RESPONSE_QUEUE*) i na dolazak poruke ponavlja postupak parsiranja. Odgovor dobiven iz poruke se potom šalje originalnom pošiljatelju.

3.2. RabbitMQ asinkroni redovi

RabbitMQ je jedan od najpopularnijih posrednika za poruke. Otvorenog je kôda, podržan na većini operacijskih sustava, a ima podršku i za mnoge programske jezike [16]. Moguće ga je pokrenuti u oblaku (engl. *cloud*) ili lokalno, a postoji i mogućnost raspodijeljene distribucije u obliku grozdova (engl. *cluster*). Također, distribucije su često u obliku slika koje se pokreću pomoću alata Docker. RabbitMQ radi na principu asinkronih redova poruka, a podržava više protokola za razmjenu poruka, kao i zaštitne protokole poput TLS-a i LDAP-a. Redovi se često koriste u slučaju potrebe za asinkronom komunikacijom između dvije komponente sustava. Za lakše praćenje redova postoje i dodatci koji pružaju korisničko sučelje u obliku internetske stranice, najčešće pokrenute na lokalnom poslužitelju. Primjer takvog sučelja je na slici 3.2. Unutar sučelja se može pratiti broj otvorenih spojeva (engl. *connection*) i kanala (engl. *channel*), broj redova, objavljene (engl. *published*) poruke, potrošene (engl. *consumed*) poruke i razne statistike o potrošnji memorije.



Slika 3.2: RabbitMQ sučelje

3.2.1. Pokretanje RabbitMQ poslužitelja pomoću Dockera

Docker je platforma koja služi za razvoj, isporuku i pokretanje aplikacija. Omogućava da se aplikacije koje se razvijaju odvoje od infrastrukture pomoću kontejnera (engl. *container*). Kontejner je jedinica koja se gradi iz Docker slike (engl. *Docker image*) i služi odvajanju aplikacije od okoline kao što je računalo, operacijski sustav ili drugi kontejner [6]. Datoteke kojima se određuju postavke i način pokretanja Docker kontejnera su *Dockerfile* i *docker_compose.yml*. *docker_compose* je fleksibilniji, prikladan u slučajevima kada je potrebno istovremeno pokrenuti više kontejnera, te će biti korišten za pokretanje RabbitMQ poslužitelja u ovom radu.

RabbitMQ poslužitelj se može pokrenuti pomoću Docker slike, odnosno kontejnera. Postoje već unaprijed napravljene slike, a jedna od njih se iskoristila za pokretanje RabbitMQ poslužitelja za potrebe sigurnosne stijene. Kôd *docker_compose* datoteke (preuzet s [15]) koja pokreće taj RabbitMQ dan je na ispisu 3.1. U datoteci se definira verzija, slika, ime kontejnera, način ponovnog pokretanja, okruženje, pristupi i postavke mreže. Linije 8 i 9 u ispisu određuju željeno korisničko ime i lozinku za pristup RabbitMQ sučelju. Njih je zbog sigurnosnih razloga potrebno promijeniti.

```
1  version: "3.7"
2  services:
3    rabbitmq:
4      image: rabbitmq:3.9.13-management-alpine
5      container_name: 'rabbitmq'
6      restart: always
7      environment:
8        - "RABBITMQ_DEFAULT_PASS=password"
9        - "RABBITMQ_DEFAULT_USER=username"
10     ports:
11       - 15672:15672
12       - 5672:5672
13     networks:
14       - rabbitmq_go_net
15
16 networks:
17   rabbitmq_go_net:
18     driver: bridge
```

Ispis 3.1 Kôd *docker_compose.yml* datoteke

Datotekama *Dockerfile* i *docker_compose*, kao i kontejnerima i slikama, upravlja se iz naredbenog retka. Najvažnije naredbe su dane na ispisu 3.2. Konkretno, pokretanje RabbitMQ poslužitelja pomoću *docker_compose* datoteke se vrši naredbama u retcima 17, 18 i 14. Nakon izvršavanja u naredbenom retku i preuzimanja zadane slike, pokrenut će se poslužitelj. Time se pokreće i sučelje, kojemu se može pristupiti pomoću internetskog preglednika na adresi *http://localhost:15672/*.

```
# pokretanje kontejnera
sudo docker build -t {app} .
sudo docker run -dp 3000:3000 {container}

# zaustavljanje i uklanjanje kontejnera i slika
sudo docker ps
sudo docker container ls
sudo docker images
sudo docker image rm {image id}
sudo docker stop {the-container-id}
sudo docker rm {the-container-id}

# pokretanje docker_compose datoteke
sudo docker-compose up

# u slucaju 'Cannot connect to the Docker daemon at unix:///var/
    run/docker.sock. Is the docker daemon running?'
sudo systemctl unmask docker
systemctl start docker
```

Ispis 3.2 Naredbe za upravljanje Dockerom iz naredbenog retka

3.2.2. RabbitMQ demonstracijski program

Sigurnosna stijena je pisana u programskom jeziku Typescript i koristi okolinu Node.js, stoga će se u tom jeziku graditi demonstracijski program kako bi se objasnio princip rada RabbitMQ asinkronih redova. Za instalaciju je potrebno dohvatiti paket *amqplib* pomoću alata *npm*.

Asinkroni redovi rade na principu proizvodnje i potrošnje poruka. Proizvođač i potrošač najprije otvaraju veze prema RabbitMQ poslužitelju. Obje strane dijele jedan kanal, za što moraju poznavati njegovo ime koje služi kao jedinstvena identifikacijska oznaka. Proizvođač objavljuje poruku u red, koji ju zabilježava i čuva. Nakon toga red obavještava potrošača da je stigla poruka, te mu je isporučuje.

Kôd razreda *RabbitMqHelper* je na ispisu 3.3. Razred se u primjeru koristi za uspostavljanje veze s poslužiteljem te za stvaranje i dohvaćanje kanala. *connectionString* koji se koristi za dobivanje *Connection* objekta je oblika *amqp://username:password@localhost:port*, gdje su korisničko ime i lozinka uzeti iz *docker_compose* datoteke, dok je pristup (engl. *port*) konstanta koju se zadaje u samom kôdu. Metoda *createChannel* stvara kanal pomoću konekcije.

```
1  import client, {Connection, Channel, ConsumeMessage} from '
       amqplib'
2
3  export class RabbitMqHelper {
4    public static async getConnection(connectionString: string):
         Promise<Connection> {
5      return await client.connect(connectionString)
6    }
7
8    public static async getChannel(connection: Connection):
         Promise<Channel> {
9      // Create a channel
10     return await connection.createChannel()
11   }
12
13 }
```

Ispis 3.3 Kôd razreda *RabbitMqHelper*

Proizvođač, odnosno razred *Publisher*, prikazan je na ispisu 3.4. Pokretanjem datoteke pomoću naredbe *npm run* izvrše se retci 37 i 38. Stvori se jedan objekt *Publisher*, koji zatim objavi deset probnih poruka. Sama metoda *publish* dohvaća spoj i kanal. Metoda *assertQueue(name: String, options?: Options.AssertQueue)* kreira red s predanim imenom i dodatnim opcijama ukoliko red tog imena ne postoji. U suprotnom se samo dohvaća referenca na već postojeći red. *durable: true* jest opcija kojom se garantira da se redovi neće uništiti pri zaustavljanju servera. No, to se isto mora učiniti i s porukama, i to opcijom *persistent: true*. Potom se poziva *sendMessages* da izvrši samo slanje poruka. *sendMessages* kao parametre prima kanal i ime reda, te šalje 10 poruka u red definiran imenom. Poruke su u ovom slučaju samo brojevi između jedan i deset. Naposlijetku se metodom *delay* simulira koristan rad i potvrđuje asinkrono ponašanje redova.

```
class Publisher {
    constructor(private readonly connectionString: string,
        private readonly queueName: string) {}

    async sendMessages(channel: Channel, queueName: string):
        Promise<void> {
        var msg: string
        for (let i = 0; i < 10; i++) {
            msg = `message \${i + 1}`
            channel.sendToQueue(queueName, Buffer.from(msg))
            console.log(`Sent \${msg}`)
            await RabbitMqHelper.delay(1000)
        }
    }

    public async publish(): Promise<void> {
        // Get a connection
        const connection: Connection = await client.connect(this
            .connectionString)

        // Create a channel
        const channel: Channel = await connection.createChannel
            ()

        // Assert queue
        await channel.assertQueue(this.queueName, {durable: true
            })

        // Send some messages to the queue
        this.sendMessages(channel, this.queueName)
    }
}
var s: Publisher = new Publisher(Constants.CONNECTION_STRING,
    Constants.QUEUE_NAME)
s.publish()
```

Ispis 3.4 Kôd u datoteci *publisher.ts*

Kôd potrošača dan je na ispisu 3.5. Potrošač se pokreće u zasebnom procesu, na primjer u drugom naredbenom retkom naredbom *npm run*. Nakon što dohvati vezu i kanal na sličan način kao i proizvođač, potrošač poziva *consume*, metodu koja pomoću *channel.ack(String)* potvrđuje prispjeće poruke. Metodu *ack* je nužno pozvati kako bi RabbitMQ poslužitelj znao da je poruka pročitana, što je bitno da bi se mogli osloboditi resursi.

```
1  class Consumer {
2      constructor(private readonly connectionString: string,
           private readonly queueName: string) {}
3
4      public async consume(): Promise<void> {
5          // consumer for the queue.
6          const consumer = (channel: Channel) => (msg:
               ConsumeMessage | null): void => {
7              if (msg) {
8                  console.log("Received " + msg.content.toString()
                       )
9                  // Acknowledge the message
10                 channel.ack(msg)
11             }
12         }
13         // Get a connection
14         const connection: Connection = await client.connect(this
               .connectionString)
15         // Create a channel
16         const channel: Channel = await connection.createChannel
               ()
17         // Start the consumer
18         channel.consume(this.queueName, consumer(channel))
19     }
20 }
21 var c: Consumer = new Consumer(Constants.CONNECTION_STRING,
       Constants.QUEUE_NAME)
22 c.consume()
```

Ispis 3.5 Kôd u datoteci *consumer.ts*

3.2.3. RabbitMQ u sigurnosnoj stijeni

U sigurnosnoj stijeni se RabbitMQ koristi na sličan način kao i u demonstracijskom primjeru, no ipak postoje i razlike. Kao što je vidljivo na ispisu 3.6, metoda *start* pokreće RabbitMQ i stvara četiri različita reda. Također se pokreću četiri različita potrošača koji čekaju da se objave poruke, a svaki od njih na prispjeće poruke poziva posebnu metodu za obradu. Metoda *getChannel* dohvaća kanal pomoću razreda *ChannelSingleton* i *ConnectionSingleton*. Oba su implementirana prema obrascu jedinstvenog objekta (engl. *singleton*), kako bi se izbjeglo stvaranje višestrukih objekata [13].

```
1  export const start = async () => {
2      var channel: Channel = await getChannel();
3      await channel.assertQueue(FILTER_REQUEST_QUEUE, {durable:
           true})
4      await channel.assertQueue(BANK_REQUEST_QUEUE, {durable: true
           })
5      await channel.assertQueue(FILTER_RESPONSE_QUEUE, {durable:
           true})
6      await channel.assertQueue(BANK_RESPONSE_QUEUE, {durable:
           true})
7
8      await channel.consume(FILTER_REQUEST_QUEUE,
           filterRequestConsumer(channel))
9      await channel.consume(BANK_REQUEST_QUEUE,
           bankRequestConsumer(channel))
10     await channel.consume(FILTER_RESPONSE_QUEUE,
           filterResponseConsumer(channel))
11     await channel.consume(BANK_RESPONSE_QUEUE,
           bankResponseConsumer(channel))
12 }
```

Ispis 3.6 Metoda *start* za pokretanje četiri reda

Nakon što su stvorena četiri reda i četiri potrošača, u proizvođačima sigurnosne stijene mogu se prihvaćati i objavljivati poruke. Postoji po jedan proizvođač u poslužitelju i klijentu te dva u dijelu za filtriranje, što odgovara prije opisanom procesu prolaska poruka kroz sigurnosnu stijenu. Ovdje će detaljno biti prikazan samo jedan smjer pro-

cesa - od poslužitelja, preko dijela za filtriranje do klijenta. Drugi smjer je gotovo identičan. Poruke koje sadrže zahtjeve moraju odgovarati sučelju *AmqpTPPRequestJsonMsg* kako bi se mogle poslati i čitati iz reda. Sučelje je definirano na ispisu 3.7. U suprotnom smjeru, kad kroz redove prolazi odgovor banke, poruka mora odgovarati sučelju *AmqpBankResponseJsonMsg*.

```
export interface AmqpTPPRequestJsonMsg {
    requestId: number,
    requestMethod: HttpMethodType,
    endpoint: string,
    headers: object,
    body: object,
    bankId: string,
}

export interface AmqpBankResponseJsonMsg {
    requestId: number,
    requestMethod: HttpMethodType,
    endpoint: string,
    responseCode: number,
    responseData: object,
    responseHeaders: object
}
```

Ispis 3.7 Sučelja *AmqpTPPRequestJsonMsg* i *AmqpBankResponseJsonMsg*

Proces u poslužitelju počinje prihvaćanjem HTTP zahtjeva kojeg šalje TPP. Zahtjev se proslijeđuje metodi *sendFilterRequestMessage*, kao što je prikazano na ispisu 3.8. Ta metoda pomoću *sendMessage* objavljuje poruku u red, a pošto je u poruku potrebno staviti objekt tipa *Buffer*, koristi se *Buffer.from(JSON.stringify(msg))*.

```
1  export const sendFilterRequestMessage = async (msg:
       AmqpTPPRequestJsonMsg) => {
2       sendMsg(FILTER_REQUEST_QUEUE, msg);
3  }
4  const sendMsg = async (queueName: string, msg:
     AmqpTPPRequestJsonMsg | AmqpBankResponseJsonMsg) => {
5      const channel = await getChannel()
6      channel.sendToQueue(queueName, Buffer.from(JSON.stringify(
         msg)), {persistent: true});
7  }
```

Ispis 3.8 Slanje poruka u red pomoću metode *sendMsg*

Poruka preko poslužitelja stiže do dijela za filtriranje. Tamo se poruka, ukoliko filter nije prepoznato zahtjev kao neispravan ili maliciozan, jednostavno proslijedi u drugi red, *BANK_REQUEST_QUEUE*. Iz njega potrošač, koji se nalazi u klijentu, čita poruku. Kôd potrošača je dan na ispisu 3.9. Poruka se nazad parsira u objekt putem *JSON.parse(msg.content.toString()) as AmqpTPPRequestJsonMsg*, te se šalje na daljnju obradu. Naposlijetku je potrebno potvrditi poruku pomoću *channel.ack(msg)*.

```
1  export const filterRequestConsumer = (channel: Channel) => (msg:
       ConsumeMessage | null): void => {
2      if (msg) {
3          const msgJSON = JSON.parse(msg.content.toString()) as
               AmqpTPPRequestJsonMsg
4          processFilterRequestJsonMsg(msgJSON)
5          channel.ack(msg)
6      }
7  }
```

Ispis 3.9 Kôd potrošača klijenta

Drugi smjer procesa počinje kad klijent dobije odgovor ASPSP-a. Zatim se poruka objavljuje u red *FILTER_RESPONSE_QUEUE*. Dio za filtriranje proslijeđuje poruku u *BANK_RESPONSE_QUEUE*, red iz kojeg čita potrošač poslužitelja. Time su završene sve radnje vezane uz proizvodnju i potrošnju poruka u sigurnosnoj stijeni.

3.3. Klijent

U klijentu se nastavlja cjelokupni proces prihvaćanja zahtjeva u sigurnosnoj stijeni. Potrošač u klijentu poziva metodu *processFilterRequestJsonMsg(msgJSON)*, koja obrađuje TPP-ov zahtjev. Tu se poruka još jednom proslijedi, ovaj put metodi *createAISRequest*. Nakon dodatnih radnji u toj metodi, dobivamo odgovor ASPSP-a koji moramo vratiti originalnom pošiljatelju, TPP-u. Stoga se u red *FILTER_RESPONSE_QUEUE* može objaviti poruka koja sadrži odgovor. To se radi pomoću metode za kreiranje povratne poruke *createAmqpBankResponseJsonMessage*, koja iz odgovora i ulazne poruke izrađuje poruku koja sadrži odgovor i prikladnog je formata za slanje u red. Za samo objavljivanje poruke zadužena je metoda *sendFilterResponseMessage*, kao što je prikazano na ispisu 3.10.

```
1 export const processBankRequestJsonMsg = async (msg:
    AmqpTPPRequestJsonMsg) => {
2     const response: AxiosResponse = await createAISRequest(msg.
        bankId, msg.headers, msg.body, msg.endpoint, msg.
        requestMethod)
3     const amqpOutputJsonMsg = createAmqpBankResponseJsonMessage(
        msg, response)
4     sendFilterResponseMessage(amqpOutputJsonMsg)
5 }
```

Ispis 3.10 Kôd metode *processBankRequestJsonMsg*

Naposlijetku postoji metoda *createAISRequest*, prikazana na ispisu 3.11. U toj metodi se pomoću HTTP klijenta *Axios* [2] šalje HTTP zahtjev na sučelje ASPSP-a. Prvo se ponovno gradi cijeli URL pomoću krajnje točke banke i drugih dostupnih podataka. U ovisnosti o parametru *requestMethod*, koji je sastavni dio poruke koju smo primili iz asinkronog reda, odlučuje se koji HTTP zahtjev je potreban. Metoda vraća odgovor koji nam daje *Axios*, odnosno odgovor ASPSP-a na poslani zahtjev. Odgovor se u obliku poruke propagira nazad do poslužitelja, koji ga šalje TPP-u.

```
1  export const createAISRequest = async (bankId: string, headers:
       any, body: any, endpoint: string, requestMethod:
       HttpMethodType): Promise<AxiosResponse> => {
2      delete headers['content-length'];
3
4      var bankUrl = getBankRouteURL(bankId) + endpoint;
5      switch (requestMethod) {
6          case HttpMethodType.GET:
7              return await axios.get(bankUrl, getRequestConfig(
                   headers, bankId)).catch(error => errorHandler(
                   error))
8          case HttpMethodType.POST:
9              return await axios.post(bankUrl, getRequestBody(
                   headers, body), getRequestConfig(headers, bankId)
                   ).catch(error => errorHandler(error))
10         case HttpMethodType.PUT:
11             return await axios.put(bankUrl, getRequestBody(
                   headers, body), getRequestConfig(headers, bankId)
                   ).catch(error => errorHandler(error))
12         case HttpMethodType.DELETE:
13             return await axios.delete(bankUrl, getRequestConfig(
                   headers, bankId)).catch(error => errorHandler(
                   error))
14         case HttpMethodType.HEAD:
15             return await axios.head(bankUrl, getRequestConfig(
                   headers, bankId)).catch(error => errorHandler(
                   error))
16         default:
17             throw 'Not implemented request method!';
18     }
19 }
```

Ispis 3.11 Kôd metode *processBankRequestJsonMsg*

4. Poslužitelj

Ovo poglavlje razrađuje strukturu i funkcionalnosti poslužitelja. Najprije se promatraju prednosti tehnologije i korištenog programskog jezika, kao i arhitektura poslužitelja. Jedna od bitnih komponenti koja se promatra je baza podataka u koju se spremaju zahtjevi koje je prihvatio poslužitelj, ali koji nisu još obrađeni. Zatim se detaljno ulazi u postupak prihvaćanja zahtjeva TPP-a, a potom i vraćanja odgovora istome.

4.1. Korištene tehnologije i arhitektura

Sigurnosna stijena pisana je u jeziku Typescript pomoću okoline Node.js. Node.js je izabran zbog brzine, skalabilnosti i podrške. Aplikacije bazirane na Node.js mogu podržati i do 40000 istovremenih spojeva u pravim uvjetima, a sama okolina je prigodna i za proširivanja po pitajnu skalabilnosti [18]. Po pitanju brzine, Node.js je bolji od drugih razmatranih okvira (engl. *framework*) kao što su Django i Flask [1].

Cijeli poslužitelj je napravljen po arhitekturalnom obrascu upravljač - pružatelj usluga - repozitorij (engl. *controller - service - repository*). Upravljač obrađuje zahtjeve i odgovore pomoću pružatelja usluga, a s bazom se komunicira putem sučelja repozitorija. To je preferirana arhitektura modernih aplikacija koja odgovara potrebama poslužitelja sigurnosne stijene.

4.2. Baza podataka za zahtjeve

Poslužitelju je potrebna baza podataka u kojoj će držati zahtjeve koje je prihvatio, a koje ostatak sigurnosne stijene još nije obradio, stoga se ne može poslati odgovor pošiljatelju. Baza podataka je u memoriji (engl. *in-memory database*), odnosno zamišljena je za privremeno čuvanje podataka. Model se sastoji od objekata *RequestObject* i *ResponseObject*, koji sadrže sve potrebne podatke o zahtjevima i odgovorima na zahtjeve. Konkretno, *RequestObject* se gradi na temelju identifikacije *id*, objekta *req*, koji je zapravo *Request* objekt *Express* biblioteke (engl. *library*), krajnje točke *endpoint* na koju je zahtjev došao, vrste HTTP zahtjeva *httpMethodType*, identifikacije banke *bankId*, te opcionalno odgovora na zahtjev, koji može biti i *undefined* ukoliko odgovor još nije stigao. *ResponseObject* se pak sastoji od identifikacije *requestId* zahtjeva za koji je odgovor vezan, vrste HTTP zahtjeva i krajnje točke (*requestMethod* i *endpoint*), zaglavlja i tijela odgovora (*responseHeaders* i *responseData*), te kôda odgovora *responseCode*.

4.2.1. Sučelje i funkcionalnosti repozitorija

U izradi baze se pratio obrazac repozitorija, stoga se interakcija vrši putem sučelja repozitorija, danog na ispisu 4.1. U sučelju su prisutne sve uobičajene metode za rukovanje podacima u bazi podataka. Prva i osnovna metoda je *addRequest*, koja gradi objekt *RequestObject* iz potrebnih podataka te ga sprema u bazu, a u slučaju greške vraća *undefined*. *addRequestObject* radi isto, samo s već gotovim *RequestObject* objektom. *getRequestById* i *popRequestForId* metode dohvaćaju *RequestObject* sa zadanom identifikacijom, a *popRequestForId* uz to i briše dohvaćeni objekt iz baze. Obje mogu vratiti *undefined* ukoliko u bazi nije pronađen zadani parametar *id*. *deleteRequestForId* briše zahtjev sa zadanom vrijednošću *id*, ukoliko postoji. Na sličan način funkcioniraju i metode za odgovore *addResponseForRequestId* i *getResponseForRequestId*. Zadnje tri metode u retcima 9, 10 i 11 služe za dodavanje, brisanje i obavještavanje kod oblikovnog obrasca promatrač (engl. *observer pattern*). Obrazac se u poslužitelju koristi za obavijesti o prispjeću odgovora za određeni zahtjev.

```
export interface RequestRepository {
    addRequest(req: Request, endpoint: string, httpMethodType:
        HttpMethodType, bankId: string, res: ResponseObject |
        undefined): RequestObject | undefined;
    addRequestObject(req: RequestObject): void;
    getRequestById(id: number): RequestObject | undefined;
    popRequestForId(id: number): RequestObject | undefined;
    deleteRequestForId(id: number): boolean;
    addResponseForRequestId(id: number, res: ResponseObject):
        void;
    getResponseForRequestId(id: number): ResponseObject |
        undefined;
    addResponseListener(requestId: number, listener:
        ResponseListener): void;
    removeResponseListenerForRequestId(requestId: number): void;
    notifyResponseListener(requestId: number): void
}
```

Ispis 4.1 Sučelje *RequestRepository*

Obrazac promatrač je iskorišten kako bi se poslužitelj mogao obavijestiti o prispjeću odgovora na zahtjev koji je spremljen u bazu. Konkretno, poslužitelj poziva *db.getRequestRepository() .addResponseListener(requestObject.getId(), listener)*, gdje je *listener* objekt koji implementira *ResponseListener* sučelje i u metodi *onResponseChanged* izvršava potrebne radnje, poput slanja pristiglog odgovora TPP-u. Samo sučelje promatrača je dano na ispisu 4.2.

```
export interface ResponseListener {
    onResponseChanged(responseObject: ResponseObject | undefined
        ): void;
}
```

Ispis 4.2 Sučelje *ResponseListener*

4.2.2. Dohvaćanje objekta *Database*

Dohvaćanje objekta koji predstavlja bazu izvodi se pomoću razreda *Database*, vidljivog na ispisu 4.3. Statičku metodu *getInstance* može se pozvati u bilo kojem dijelu kôda koji ima referencu na sam razred *Database*. Baza je jedinstveni objekt koji se instancira pri prvom zvanju metode *getInstance*. Pristup repozitoriju je omogućem putem metode *getRequestRepository*, koja također koristi obrazac jedinstvenog objekta.

```
1  export class Database {
2      private static requestRepository: RequestRepository;
3      private static instance: Database | undefined = undefined;
4
5      private constructor() {
6          Database.requestRepository = RequestRepositoryImpl.
               getInstance();
7      }
8
9      public static getInstance(): Database {
10         if(this.instance === undefined) {
11             this.instance = new Database();
12         }
13         return this.instance;
14     }
15
16     public getRequestRepository() {
17         return Database.requestRepository;
18     }
19 }
```

Ispis 4.3 Sučelje *ResponseListener*

4.3. Prihvaćanje i obrada zahtjeva

Poslužitelj je dužan imati sučelje koje odgovara aplikacijskom programskom sučelju ASPSP-a. Dakle, potrebno je imati krajnje točke kakve ima i banka. Pošto je sigurnosna stijena posrednik, potrebno je pokrenuti posrednički poslužitelj. Za to je izabrana lokalna distribucija putem biblioteke *Express*.

4.3.1. Krajnje točke

Uz pomoć *Express* biblioteke pokreće se lokalni poslužitelj i dodaju se potrebne krajnje točke. Dvije od njih su dane kao primjer u ispisu 4.4. Jedna prihvaća *POST* zahtjeve na krajnju točku */consents*, a druga *GET* zahtjeve na */accounts*. Nakon uklanjanja prefiksa specifičnog za pojedinu banku i postavljanja odgovarajućeg tipa HTTP zahtjeva, obje proslijeđuju obradu zahtjeva metodi *sendRequest*.

```
1  app.post(Endpoints.POST_CONSENTS_ENDPOINT, async (req: Request,
       res: Response): Promise<void> => {
2      var endpoint = removePrefix(Endpoints.POST_CONSENTS_ENDPOINT
           );
3      var httpMethodType = HttpMethodType.POST;
4      sendRequest(db, req, res, endpoint, httpMethodType);
5  });
6
7  app.get(Endpoints.GET_ACCOUNTS_ENDPOINT, async (req: Request,
       res: Response): Promise<void> => {
8      var endpoint = removePrefix(Endpoints.GET_ACCOUNTS_ENDPOINT)
           ;
9      var httpMethodType = HttpMethodType.GET;
10     sendRequest(db, req, res, endpoint, httpMethodType);
11 });
```

Ispis 4.4 Sučelje *ResponseListener*

4.3.2. Upravljač i pružatelj usluga

Detalji metode *sendRequest* prikazani su na ispisu 4.5. Pri obradi se najprije dohvaća parametar zaglavlja *bankId*. U slučaju neispravnog slanja tog parametra, zahtjev se ne propušta dalje, nego se poziva dio kôda za rukovanje pogreškama, konkretno metoda *sendError*. Ako je zahtjev validan, dodaje se u bazu podataka pomoću sučelja repozitorija, kako je objašnjeno u prethodnom potpoglavlju. U slučaju uspješnog dodavanja, poziva se metoda zadužena za daljnju obradu. Uz to se dodaje i promatrač, čiji će detalji biti prikazani u sljedećem potpoglavlju.

```
1  export const sendRequest = async (db: Database, req: Request,
       res: Response, endpoint: string, httpMethodType:
       HttpMethodType,
2      controllerFunction: (requestObject: RequestObject) =>
         Promise<void> = AISController.sendRequest) => {
3
4      var bankId: string | string[] | undefined = req.headers[
         bankIdString];
5
6      switch (typeof bankId) {
7          case "string":
8              delete req.headers[bankIdString];
9
10             var requestObject: RequestObject | undefined = db.
                   getRequestRepository().addRequest(req, endpoint,
                   httpMethodType, bankId, undefined);
11
12             if (requestObject === undefined) {
13                 handleError();
14             } else {
15                 controllerFunction(requestObject);
16                 addListener(db, requestObject, res);
17             }
18
19             break;
20         case "undefined":
21         default:
22             sendError(res, "Incorrect bank-id!");
23     }
24 }
```

Ispis 4.5 Kôd metode *sendRequest*

Metoda *sendRequest* u pružatelju usluga je prikazana na ispisu 4.6. Razred *RequestObject* sadrži metodu *toAmqpInputMsg*, koja iz elemenata objekta stvara cjelovitu poruku koja je instanca sučelja *AmqpTPPRequestJsonMsg*. Nakon toga, poruka se šalje u RabbitMQ red pomoću za to definirane metode *sendFilterRequestMessage*. Time završava obrada zahtjeva u poslužitelju sigurnosne stijene.

```
export const sendRequest = async (req: RequestObject) => {
    var msg = req.toAmqpInputMsg();
    sendFilterRequestMessage(msg);
}
```

Ispis 4.6 Kôd metode *sendRequest* u pružatelju usluga

4.4. Vraćanje odgovora pošiljatelju

Nakon što dio za filtriranje i klijent odrade svoj dio posla, odgovor ASPSP-a je spremljen kao poruka u redu *BANK_RESPONSE_QUEUE*. Potrošač koji čita iz tog reda parsira poruku, te ju šalje na obradu u metodu *processBankResponseJsonMsg*, danu na ispisu 4.7. Dobivena poruka je instanca sučelja *AmqpBankResponseJsonMsg*. Nakon ispisa, poruka koja sadrži odgovor ASPSP-a se sprema u bazu podataka.

```
export const processBankResponseJsonMsg = async (msg:
    AmqpBankResponseJsonMsg) => {
    logResponse(msg);
    storeResponse(msg);
}

const storeResponse = async (msg: AmqpBankResponseJsonMsg) => {
    var db = Database.getInstance();
    var res: ResponseObject = new ResponseObject(msg.requestId,
        msg.requestMethod, msg.endpoint, msg.responseHeaders, msg
        .responseData, msg.responseCode);
    db.getRequestRepository().addResponseForRequestId(msg.
        requestId, res);
}
```

Ispis 4.7 Kôd potrošača za red *BANK_RESPONSE_QUEUE*

Spremanjem u bazu putem metode *addResponseForRequestId* aktivira se promatrač dan na ispisu 4.8. Ako nije došlo do greške, odgovor se šalje pomoću *Express* objekta *Response*. Taj objekt ne može primiti cjelokupno zaglavlje odgovora odjednom, stoga se iteracijom po zaglavlju postavlja svaki parametar zasebno. Naposlijetku, postavljanjem kôda odgovora i slanjem JSON-a završava proces vraćanja odgovora ASPSP-a TPP pošiljatelju.

```
1  export const addListener = function (db: Database, requestObject
       : RequestObject, res: Response) {
2      var listener = {
3          onResponseChanged(responseObject: ResponseObject |
               undefined): void {
4              if (responseObject === undefined) {
5                  handleError("Listener error! Response object is
                       undefined. Should not happen.");
6              } else {
7                  // set headers
8                  var headers = JSON.parse(JSON.stringify(
                       responseObject.getResponseHeaders()));
9                  for (let member in headers) {
10                     res.setHeader(member, headers[member]);
11                 }
12
13                 // set status and send as json
14                 res.status(responseObject.getResponseCode()).
                       json(responseObject.getResponseData());
15             }
16         }
17     }
18 }
```

Ispis 4.8 Promatrač u poslužitelju

5. Testiranje

U ovom poglavlju se govori o testiranju poslužitelja, a time i cijele sigurnosne stijene. Prvo se objašnjava proces pokretanja poslužitelja uz pomoć Node.js okoline. Poslije toga se poslužitelj testira slanjem ispravnih i neispravnih zahtjeva pomoću Python skripte.

5.1. Pokretanje poslužitelja

Pokretanjem poslužitelja pokreće se i cjelokupni skup mikroservisa sigurnosne stijene. Prvi dio postupka je pokretanje RabbitMQ poslužitelja u naredbenom retku pomoću naredbe *sudo docker-compose up*, kako je opisano u prethodnim poglavljima. Zatim se pokreće sam poslužitelj u drugom naredbenom retku. Postoje tri skripte definirane u *package.json* postavkama, a dva su načina pokretnja. Prva je kombinacija naredbi koje pokreću skripte *npm run build* i *npm run server*. Pri tome prva naredba prevodi Typescript kôd u Javascript, a druga pokreće ulaznu točku, *index.js*. Drugi način pokretanja je u razvojnom okruženju, pomoću *npm run dev*. Ta skripta koristi alate *concurrently* [5] i *nodemon* [14] kako bi se pri svakom spremanju poslužitelj automatski ponovno pokrenuo. Osim u naredbenom retku, moguće je i pokrenuti kôd direktno iz IDE-a po izboru.

Poslužitelj se pokreće lokalno na `http://localhost:8000`. Nakon što se pokrene na jedan od opisanih načina, moguće je testirati jednostavne funkcionalnosti slanjem zahtjeva na postojeću krajnju točku. U slučaju da poslužitelj ispiše sadržaj zahtjeva, a zatim i sadržaj odgovora u naredbeni redak, znači da je zahtjev legalan i odgovor je vraćen pošiljatelju.

5.2. Skripta

Testiranje poslužitelja izvršava se pomoću Python skripte. Ona je zamišljena kao lako izmjenjiv niz zahtjeva koji čine logički slijed i vrše neku radnju. Za testnu banku se uzela Erste zaštićena okolina (engl. *sandbox*), kojoj se mogu slati svi zahtjevi kao i na pravu banku, a podaci koje okolina vraća su probni. Primjer koji će služiti za testiranje je dohvaćanje računa nekog korisnika. Predradnje za dohvaćanje računa su *POST* zahtjev na krajnju točku */consents*, a zatim *POST* zahtjev na */consents/:consentId/authorisations/:authorisationId/token*. Prvi navedeni zahtjev stvara *Consent-ID*, a drugi token za pristup (engl. *access token*). Bez ta dva podatka nije moguće pristupati resursima poput računa. Nakon prethodna dva koraka, moguće je dohvatiti račune HTTP pozivom *GET* na krajnju točku */accounts*.

Prvi test u skripti je slanje tri prethodno opisana zahtjeva u pravilnom redoslijedu, s dobro postavljenim parametrima i pravilnim zaglavljem i tijelom. Kad se pokrene skripta, na poslužitelju se ispisuju svi pristigli zahtjevi i njihovi odgovarajući odgovori. Skraćeni prijepis iz naredbenog retka dan je na ispisu 5.1. Na takav slijed zahtjeva dobiven je prikladan odgovor u kojem se nalaze svi računi korisnika asociranog s jedinstvenom identifikacijom *PSU-ID*. *PSU-ID* se mora poslati kao parametar zaglavlja u zadnjem zahtjevu prema krajnjoj točki */accounts*.

```
POST request on "/consents"
"requestId": 0,
"requestMethod": 1,
"endpoint": "/consents",
"headers": ... ,
"body": ... ,
"bankId": "erste_sandbox"
...
Response
endpoint: /consents
...

POST request on "/consents/ba9d90c4-61e7-4013-bf95-6ac857d1b2af/
    authorisations/edc7fab0-fa6a-4fd1-a982-148469668a52/token"
"requestId": 1,
"requestMethod": 1,
...
Response
endpoint: /consents
...

GET request on "/accounts"
"requestId": 2,
"requestMethod": 0,
...
Response
endpoint: /accounts
request id: 2
request method: 0
response headers: ...
response data: {
    "accounts": [
        ...
    ]
}
response code: 200
```

Ispis 5.1 Primljeni zahtjevi i odgovori u poslužitelju

Drugi test je izvršavan na istom slijedu zahtjeva, samo što je izmijenjen *PSU-ID* na neki nepostojeći. U tom slučaju se svi zahtjevi uobičajeno izvedu, no kao odgovor se dobije JSON *"accounts": []* . To znači da ne postoje računi asocirani s pogrešnim *PSU-ID*, što je i predviđeno ponašanje.

Sljedeći test uključuje slanje istog slijeda zahtjeva, pri čemu su prva dva u potpunosti točna, dok je zadnji poslan na nepostojeću krajnju točku, na primjer */acounts* umjesto *accounts*. Ideja je da se takav tipfeler uistinu može dogoditi pri slanju zahtjeva. Skripta dobiva odgovor prikazan na ispisu 5.2. Vidljivo je da poslužitelj ne prihvaća zahtjev na nepostojeću krajnju točku.

```
1 <html lang="en">
2 <head>
3 <meta charset="utf-8">
4 <title>Error</title>
5 </head>
6 <body>
7 <pre>Cannot GET /v1/psd2-aisp/acounts</pre>
8 </body>
9 </html>
```

Ispis 5.2 Primljeni odgovor u skripti

Još jedan test je odmicanje od zamišljenog toka zahtjeva, na primjer pokušaj dohvaćanja računa bez da se stvorio token za pristup, odnosno izvršavanje prvog i trećeg HTTP zahtjeva iz očekivanog slijeda. U takvom slučaju, dobiva se greška *TOKEN_MISSING* s kôdom odgovora 403, što označuje pokušaj dohvaćanja zabranjenog resursa.

Provedeno je nekoliko sličnih testiranja kojima je zaključeno da točno definirani zahtjevi u pravilnom redoslijedu vraćaju očekivane odgovore. S druge strane, kod promjene parametara u pogrešne ili zamjene redoslijeda zahtjeva, vraćaju se greške slične već prikazanima. Stoga se zaključuje da poslužitelj funkcionira kako je predviđeno.

6. Zaključak

U ovom radu je implementiran poslužitelj za PSD2 sigurnosnu stijenu. Najprije je objašnjena važnost i opisane specifičnosti PSD2 direktive te konkretnog standarda NextGenPSD2 koji se koristi u hrvatskim bankama. Zatim se proučila sigurnosna stijena, njene funkcionalnosti, arhitektura i komponente od kojih se sastoji. Jedna od glavnih komponenti je poslužitelj, čiji su izgled, zadaće i upotreba detaljno opisani. Naposlijetku je poslužitelj testiran pomoću Python skripte.

Kibernetička sigurnost jedna je od najvažnijih tema u današnjem bankarstvu. Uvođenjem PSD2 direktive hrvatske banke prisiljene su otvoriti svoja aplikacijska programska sučelja, što može dovesti do ranjivosti. Izgrađena sigurnosna stijena, a time i njen poslužitelj, mogu spriječiti neke napade na API-je banaka, čineći ih sigurnijim. Stoga su rezultati ovog rada zadovoljavajući.

Postoji mnogo vrsta napada i neotkrivenih ranjivosti, od kojih su samo neki uzeti u obzir pri gradnji sigurnosne stijene. Ostali napadi bi se mogli proučiti i isprobati pomoću dodatnih testiranja, na primjer proširenjem Python skripte. Također, poslužitelj se mora modificirati u ovisnosti o distribuciji sigurnosne stijene. Na primjer, moguće je ponuditi više rješenja za sigurnosnu stijenu, od kojih je jedno smješteno u oblaku, a drugo direktno implementirano u API-ju ASPSP-a. Zato se u budućnosti poslužitelj može nadograditi kako bi još bolje štitio sučelje ASPSP-a i bio prikladan za više vrsta distribucija.

7. LITERATURA

[1] Abel Avram. *Comparing the Performance of Various Web Frameworks*, pristupljeno 15.6.2022. URL https://www.infoq.com/news/2014/05/benchmark-web-framework/.

[2] Axios. *Promise based HTTP client for the browser and node.js*, pristupljeno 12.6.2022. URL https://axios-http.com/.

[3] bankIO. *NextGenPSD2*, pristupljeno 3.6.2022. URL https://bankio.at/openbanking/knowledge-base/NextGenPSD2/.

[4] BBVA. *PSD2 Explained: What is it and why does it matter?*, pristupljeno 1.6.2022. URL https://www.bbva.com/en/everything-need-know-psd2/.

[5] concurrently. *Concurrently TS*, pristupljeno 14.6.2022. URL https://www.npmjs.com/package/concurrently.

[6] Docker. *Docker overview*, pristupljeno 12.6.2022. URL https://docs.docker.com/get-started/overview/.

[7] European Union. *Directive (EU) 2015/2366 of the European Parliament and of the Council*, pristupljeno 1.6.2022. URL https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A02015L2366-20151223.

[8] Sead Fadilpašić. *Banks spending three times more on cyber security*, pristupljeno 1.6.2022. URL https://www.itproportal.com/news/banks-spending-three-times-more-on-cyber-security/.

[9] FinExtra. *Berlin Group publishes NextGenPSD2 framework*, pristupljeno 3.6.2022.

[10] Tarik Karamehmedović Fran Vrdoljak, Antonio Špoljar. *psd2-diplomski*, pristupljeno 25.6.2022. URL https://gitlab.com/franvrdoljak/psd2-dimplomski.

[11] Brian Gaynor. *Payment Services Directive 2 – an overview*. JP Morgan, pristupljeno 8.6.2022. URL https://www.jpmorgan.com/europe/merchant-services/insights/PSD2-all-you-need-to-know.

[12] GIU HUB. *PSD2 Open API*, pristupljeno 3.6.2022. URL https://www.hub.hr/hr/PSD2-Open-Api-hr.

[13] Refactoring guru. *Singleton in TypeScript*, pristupljeno 13.6.2022. URL https://refactoring.guru/design-patterns/singleton/typescript/example.

[14] nodemon. *nodemon TS*, pristupljeno 14.6.2022. URL https://www.npmjs.com/package/nodemon.

[15] Jamie Pennell. *How to Talk to a RabbitMQ Instance with Node.js and TypeScript*, pristupljeno 12.6.2022. URL https://javascript.plainenglish.io/how-to-talk-to-a-rabbitmq-instance-with-node-js-and-typescript-92

[16] RabbitMQ. *RabbitMQ Features*, pristupljeno 12.6.2022. URL https://www.rabbitmq.com/#features.

[17] Revolut. *Revolut - One app, all things money*, pristupljeno 17.6.2022. URL https://www.revolut.com/.

[18] Davit Vardanyan. *How Many Requests Can a Real-World Node.js Server-Side App Handle?*, pristupljeno 15.6.2022. URL https://javascript.plainenglish.io/how-many-requests-can-handle-a-real-world-nodejs-server-side-appl

[19] Bob Violino. *Cybersecurity spending puts new emphasis on detection and response*, pristupljeno 9.6.2022. URL https://www.information-management.com/news/cybersecurity-spending-puts-new-emphasis-on-detection-and-respons

Implementacija PSD2 poslužitelja

Sažetak

Cilj ovog rada je gradnja poslužitelja za posredničku sigurnosnu stijenu koja štiti PSD2 sučelja banaka. Prvi korak rada je proučavanje direktive PSD2 te standarda NextGenPSD2 koji je prihvaćen u hrvatskim bankama. Nakon opisa funkcionalnosti, arhitekture i pojedinih dijelova sigurnosne stijene, počinje se s konstrukcijom samog poslužitelja. Najprije se ulazi u detalje vezane uz funkcionalnosti i komponente poslužitelja. Prikazuje se rukovanje bazom podataka u koju poslužitelj sprema zahtjeve kako bi kasnije, nakon daljnje obrade zahtjeva u sigurnosnoj stijeni, mogao poslati odgovor. Implementira se prihvaćanje zahtjeva i slanje odgovora, što su dva glavna zadatka poslužitelja. Naposlijetku se putem skripte provodi testiranje poslužitelja, a time i cijele sigurnosne stijene, kako bi se potvrdilo da rade kako je zamišljeno.
Ključne riječi: sigurnosna stijena, NextGenPSD2, RabbitMQ, sigurnost, Docker, Node.js

PSD2 server implementation

Abstract

The aim of this study is to build a server for PSD2 proxy firewall, which protects bank APIs. The first step is to study the PSD2 directive, as well as the NextGenPSD2 standard, which is an accepted standard in Croatian banks. After the description of functionalities, architecture and certain parts of firewall, the study follows the construction of the server itself. Firstly, the details on server functionality and components are fleshed out. The study then explains the usage of server's request database, which is needed for storing the requests while the rest of the firewall processes them and prepares the response. The two main jobs of the server - accepting the requests and sending the responses - are implemented as well. In the end, the server and firewall are tested with a script to verify their functionalities.

Keywords: firewall, NextGenPSD2, RabbitMQ, security, Docker, Node.js