Web aplikacija Todo liste implementirane pomoću Rust programskog jezika

Fačko, Dominik

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Stručni studij

WEB APLIKACIJA TODO LISTE IMPLEMENTIRANE POMOĆU RUST PROGRAMSKOG JEZIKA

Završni rad

Dominik Fačko

Osijek, 2021.

SADRŽAJ

1. UVOD 1

1.1 Zadatak i ciljevi rada 1

2 SLIČNE APLIKACIJE 2

3. O TEHNOLOGIJAMA 3

3.1. Što je Rust? 3

3.2 Actix 4

3.3. Postman 4

3.4. Docker 5

3.5. DBeaver 5

3.6. Baza podataka 5

3.7 JSON Web Token 6

4. APLIKACIJE TODO LISTE NAPISANA U RUST-U 8

4.1. Struktura aplikacije 8

4.2. Primjer korištenja 18

***5.* ZAKLJUČAK** 27

LITERATURA 28

SAŽETAK 29

ABSTRACT 30

ŽIVOTOPIS 31

PRILOZI 32

1. UVOD

Nagli porast broja raznih tehnologija, novih aplikacija i broja korisnika dovelo je do novog načina razvoja i načina distribucije aplikacija, olakšanog procesa verzioniranja i održavanja aplikacija, a samim time pristupanje i korištenje tih aplikacija je manje zahtjevna za korisnike.

Za razliku od uobičajenih aplikacija koje se nalaze lokalno kod korisnika, u memoriji računala na kojem su instalirana, web aplikacije su instalirane i pokrenute na web serverima. S time se znatno smanjuju zahtjevi i kriteriji na računala koja žele pristupiti aplikacijama.

U ovoj temi govorit će se o tome kako je moguće izgraditi takvu web aplikaciju, zahtjevima pri izgradnji kao što su provjera podataka (engl. *data validation),* i provjera zahtjeva (engl. *request validation).*

Ovaj završni rad bit će raspisan u više dijelova. U drugom poglavlju pokazat će se aplikacije izgrađene na sličan način u području web aplikacija. U trećem poglavlju „*O tehnologijama*“ bit će objašnjene tehnologije korištene za izradu ove aplikacije a u četvrtom poglavlju bit će objašnjena sama struktura aplikacije i način kako se koristiti ovom aplikacijom. Glavni osvrt ovog završnog rada bit će na izradu web aplikacije u Rust programskom jeziku.

1.1 Zadatak i ciljevi rada

Prikaz detaljne implementacije različitih funkcionalnosti unutar web aplikacije TODO liste napisane u Rust programskom jeziku, od konekcija na bazu do upravljanja s pogreškama. Student će detaljno razložiti pojedinu funkcionalnost aplikacije te staviti naglasak kako se ona detaljno implementira u Rust programskom jeziku. Student će kreirati funkcionalnu aplikaciju te detaljno objasniti funkcioniranje backend dijela aplikacije.

2 SLIČNE APLIKACIJE

Uz ovaj završni rad izrađeno je već nekoliko drugih završnih radova koji su slični u nekom svom aspektu npr. po korištenim tehnologijama, problemu koji se probao riješit.

- „*Web sustav za upravljanje projektima*“, Matej Dragun, FERIT [1].
- „*Web aplikacija za objavu oglasa*“, Dragana Udovičić, FERIT [2].
- „*Youtube“* [3].

U radu „*Web sustav za upravljanje projektima*“ vidi se sličnost u ideji samog projekta. Web aplikacija koja omogućuje korisnicima upravljanje nekakvim resursima, u slučaju tog rada su to projekti, isto tako vidi se sličnost u korištenoj bazi podataka, obje baze su SQL baze. U sljedećem radu „*Web aplikacija za objavu oglasa*“ vidi se sličnost isto u tome što se u oba rada govori o web aplikacijama, te obje aplikacije podržavaju CRUD operacija nad unesenim podatcima. Oba gore navedena rada su slična ovom u više aspekta, oba se temelje na unosu podataka od strane korisnika, procesuiranje tih podataka, pohrana i njihovo dohvaćanje. Jedno od ključnih razlika je da se podatci u TODO listi svi privatni, te ne postoji mogućnost da korisnici međusobno dijele, ili na bilo koji način vide podatke. Svi uneseni podatci su privatni. „*Youtube“* je poznata platforma za online dijeljenje i pregled video sadržaja, a sličnost između ovog rada i „*Youtube“*-a je u play-list funkcionalnosti na „*Youtube“*-u, na način da korisnici mogu kreirati svoje play-liste, slično kao TODO liste u ovom radu, i isto tako mogu dodavati i brisati elemente s tih listi, gdje su ti elementi nekakve pjesme ili video sadržaj na „*Youtube*“-u a u ovoj aplikaciji to su TODO stavke.

3. O TEHNOLOGIJAMA

Tehnologije korištene u ovom projektu su: Rust programski jezik kao glavni alat za izradu web aplikacije, Postman kao alat za komuniciranje s web aplikacijom, u ovom radu sa serverom, Docker kao alat koji je u ovom radu korišten za podizanje baze podataka u odvojenom okruženju (engl. *environment)* i DBeaver kao glavni alat za pristup bazi.

3.1. Što je Rust?

Rust je programski jezik niske razine (engl. *low-level systems programming language)*. Rust kao programski jezik prvo se službeno pojavio 2010. godine od strane Mozilla-e što ga čini vrlo mladim. Glavni cilj Rust-a kao programskog jezika je biti memorijski siguran (engl. *memory safe)*, nitno siguran (engl. *thead safe)*.

Kako bi se to postiglo Rust uvodi koncepte kao što su vlasništvo (engl. *ownership)*, posuđivanje (engl. *borrowing)*, životni vijek (engl. *lifetimes)* i zbog toga većina pogrešaka (engl. *error*) u Rust-u se dogodi za vrijeme sastavljanja (engl. *compile time*) nego za vrijeme izvršavanja (engl. *runtime*) [4].

Vlasništvo i životni vijek u Rust-u nam govori koji dio programa ili aplikacije je vlasnik kojeg dijela memorije, tj. u kojem dijelu memorije živi koja varijabla. Zbog toga u Rustu nema potrebe za vođenjem brige o visećim pokazivačima (engl. *dangling pointers)* i Rust nema sakupljača smeća (engl. *garbage collector*).

Rust sastavljač (engl. *compiler*) vodi brigu o tome koja funkcija ili koji kontekst trenutno ima vlasništvo nad kojim podatcima u bilo kojem trenutku i zbog toga nigdje u kôdu ne može se pristupiti podatcima koji nisu unutar trenutnog konteksta. Nakon što kontekst ili određena funkcija završi, svi podatci unutar nje su obrisani i memorija oslobođena.

Ovdje se dolazi do posuđivanja u Rust-u. Pošto su svi podatci i memorija usko vezani za funkciju ili nekakav opseg u kojem se trenutno nalaze, a često dolazi do potrebe za tim podatcima i izvan tih opsega treba postojati način kako doći do tih podataka tj. posuditi ih iz opsega u kojem se one nalaze.

Posuđivanje podataka u Rust-u može se dogoditi na tri različita načina, a to su:

1. Promjenjivo (engl. *mutable*)
2. Nepromjenjivo (engl. *immutable*)

Promjenjivo posuđivanje znači da funkcija koja je posudila podatak, može ga promijeniti, ali, nakon završetka s radom mora taj podatak i vratiti tamo odakle je posuđen. Jedno pravilo kod ovakvog posuđivanja podataka je da, ako je jedan podatak posuđen od roditeljske funkcije nijedna druga funkcija ga ne može posuditi dok on nije vraćen.

Nepromjenjivo posuđivanje znači da se posuđeni podatak može samo čitati bez bilo kakvog mijenjanja, ovdje, kod promjenjivog posuđivanja, prethodno pravilo ne vrijedi, što znači da se jedan podatak može posuditi ili jednom promjenjivo ili beskonačno puta nepromjenjivo [5].

3.2 Actix

Actix je web okvir za Rust i on je glavni paket korišten za kreiranje samog servera ove aplikacije. Actix je najpoznatiji i najkorišteniji web okvir za rust uz drugi okvir zvan Rocket. U ovom radu odlučeno je koristiti Actix iz razloga što on, za razliku od Rocketa, ne sadržava gotova rješenja za česte probleme web developmenta.

Actix je brži od Rocketa, ali je i noviji okvir te zbog toga puno funkcionalnosti, koje Rocket pruža kao gotova rješenja, u Actixu se treba implementirati koristeći pakete trećih strana (engl. *third party packages*) [6].

3.3. Postman

Postman je alat koji uz mnoge druge stvari služi za interakciju sa sučeljima za programiranje aplikacija (engl. *Application Programming Interface – API)* koja su u skladu s reprezentativnim prijenosom stanja (engl. *Representational state transfer – REST)*. Takva sučelja nazivamo „RESTful API“. Za izradu ovog projekta korišten je kao glavni alat za interakciju s aplikacijom jer sama aplikacija nema prednji kraj (engl. *front-end*). Komunikacija s aplikacijom kroz Postman odvijala se preko tzv. „CRUD“ zahtjeva (engl. *Request*) koji se sastoje od tzv. „*JSON*“ objekata i dodatnih informacija sadržanim u zaglavljima (engl. *Header*) zahtjeva [7].

3.4. Docker

Docker je otvorena platforma za razvoj, dijeljenje i pokretanje aplikacija. Docker omogućuje razdvajanje aplikacije od infrastrukture sustava, što znači da pomoću Docker slike (engl *Docker image*) možemo pokrenuti aplikaciju na bilo kojem sustavu. Docker kontejner (engl. *Docker container*) je jedinica koja u sebi sadrži sav potreban kôd i ostale ovisnosti aplikacije kako bi ona radila. U ovom projektu korišten je Docker kontejner koji u sebi sadrži bazu Postgres [8].

3.5. DBeaver

DBeaver je alat za interakciju s bazama podataka podržan na više operacijskih sustava kao što su Linux, Mac i Windows. Uz prikaz informacija o povezanoj bazi, pruža i grafičko sučelje za prikaz sadržaja baze. Podržava većinu poznatijih baza podataka kao što su MySQL, PostgreSQL, Oracle, MS Access, Firebird, Apache Hive itd. Osim grafičkog prikaza podataka iz baze, DBeaver podržava pisanje i izvršavanje skripti za podržane baze te je jako koristan alat u bilo kakvom razvojnom procesu [9] .

3.6. Baza podataka

Za izradu ovog projekta korištena je baza PostgreSQL. PostgreSQL je sustav otvorenog kôda (engl. *Open-source*) za upravljanje objektno-relacijskim bazama podataka (engl. *Object-relational*). PostgreSQL podržava relacijsko i ne relacijsko ispitivanje (engl. *Querying*) baze. PostgreSQL je jedna od najčešće korištenih baza podataka zbog svoje efikasnosti i skalabilnosti te koristi se kao glavna baza u raznim projektima. PostgreSQL podupire razne programske jezike kao što su Python, Java, C#, C/C++, Ruby, JavaScript (Node.js) i Go [10].

Prije izrade bilo kakvog projekta potrebno je odlučiti ima li potrebe da baza podataka koja će se koristiti bude relacijska ili ne. Glavna razlika između relacijskih i ne relacijskih baza je to što su podatci u relacijskim bazama pohranjeni u tabličnoj strukturi sa stupcima, koji se još nazivaju i atributima, i redovima, koji zapravo predstavljaju jedan cijeli zapis (engl. *Entry/* slično kao u proračunskoj tablici (engl. *Spreadsheet*). Ne relacijske baze podataka temelje se na pravilnom

izboru modela skladištenja (engl. *Storage model*) ovisno o tome koji od tih modela najbolje odgovara podatcima koje će biti spremljene u bazu.

Dok se u ne relacijskim bazama podatci u bazi nazivaju dokumentima, u relacijskim bazama to su entiteti. Entitet predstavlja bilo kakav podatak spremljen u bazu. Entiteti su najčešće međusobno povezani, ta poveznica među entitetima naziva se relacija, te tako postoji više vrsta relacija:

1. Jedan prema jedan.
2. Jedan prema više.
3. Vise prema više.

Osim međusobnih relacija, druga važna stavka koju entiteti mogu imati su primarni ključ PK (engl. *Primary key*), i vanjski ključ FK (engl. *Foreign key*). Primarni ključ je stupac ili više stupaca u tablici koje jedinstveno određuju/identificiraju red u tablici. Pošto je primarni ključ jedinstveni identifikator on se može ponoviti samo jednom. Strani ključ u tablici je stupac čija vrijednost odgovara primarnom ključu iz jedne druge tablice, u jednoj tablici može postojati više stranih ključeva, te oni služe za povezivanje te tablice s nekom drugom.

Relacija jedan prema jedan (1 – 1) znači da glavni ključ jednog entiteta može biti vezan za samo jedan strani ključ jednog drugog entiteta (npr. Osoba može imati samo jednu osobnu iskaznicu).

Relacija jedan prema više (1 – N) znači da glavni ključ jednog entiteta može biti vezan za više stranih ključeva (npr. Osoba može imati više auta u vlasništvu, ali određeni auto ima samo točno jednog vlasnika).

Za relaciju više prema više potrebno je dodavanje jedne među tablice u kojoj su primarni ključevi povezanih entiteta. Tako se odnos više prema više svodi na jedan prema više (npr. Vlasnik može biti u vlasništvu više kompanija, a svaka od tih kompanija može imati više od jednog vlasnika).

3.7 JSON Web Token

JSON Web Token je standard koji opisuje način kako se među više stranaka podatci mogu sigurno dijeliti u *„JSON“* obliku. Podatci koji su na ovaj način poslani i primljeni mogu se

verificirati te im se može vjerovati da su ispravni i da nisu promijenjeni na putu do odredišta, jer su svi podatci digitalno potpisani od stranke koja ih šalje.

JSON Web Token-i mogu se potpisati i provjeriti istim ključem, ovakav pristup korišten je u ovoj aplikaciji za provjeru autentičnosti HTTP zahtjeva, na ovaj način podatci u samom „*jwt*“-u su vidljivi svima koji ga presretnu na putu, ali se ne mogu promijeniti jer digitalni potpis neće odgovarati sadržaju. JSON Web Token-i isto tako mogu se kriptirati s parom javnih/privatnih ključeva. Na ovaj način sadržaj „*jwt*“-a nitko ne može vidjeti, ako nema javni ključ za dešifriranje [11].

Sam „*jwt*“ sastoji se od tri djela tj. tri zasebna „*JSON*“ objekata gdje se svaki zasebno kodira u , ti dijelovi su:

- Zaglavlje (engl. *header*)
- Nosivost (engl. *payload*)
- Potpis (engl. *signature*)

Zaglavlje sastoji se od dva značajna ključa, jedan se odnosi na tip samog token-a, u ovom slučaju to je „*jwt*“, a drugi ključ odnosi se na algoritam koji se koristi za potpisivanje, u ovom projektu to je bio HS256. Ovaj dio token-a kodira se u formatu „*Base64Url*“.

Nosivost je sadržaj token-a, u ovom djelu su svi podatci koji se žele podijeliti te koje će druga strana verificirati. Postoji nekoliko unaprijed definiranih ključeva koji se mogu koristiti u ovom dijelu token-a, od kojih su najčešći *„iss“* koji govori tko je izdavatelj token-a, „*sub*“ koji govori o predmetu token-a i „*exp*“ koji govori koliko dugo token vrijedi. Mogu se dodati i još prilagođeni (engl. *custom*) ključevi kako bi se prenio dodatan sadržaj. Ovaj dio tokena kodira se u formatu „*Base64Url*“.

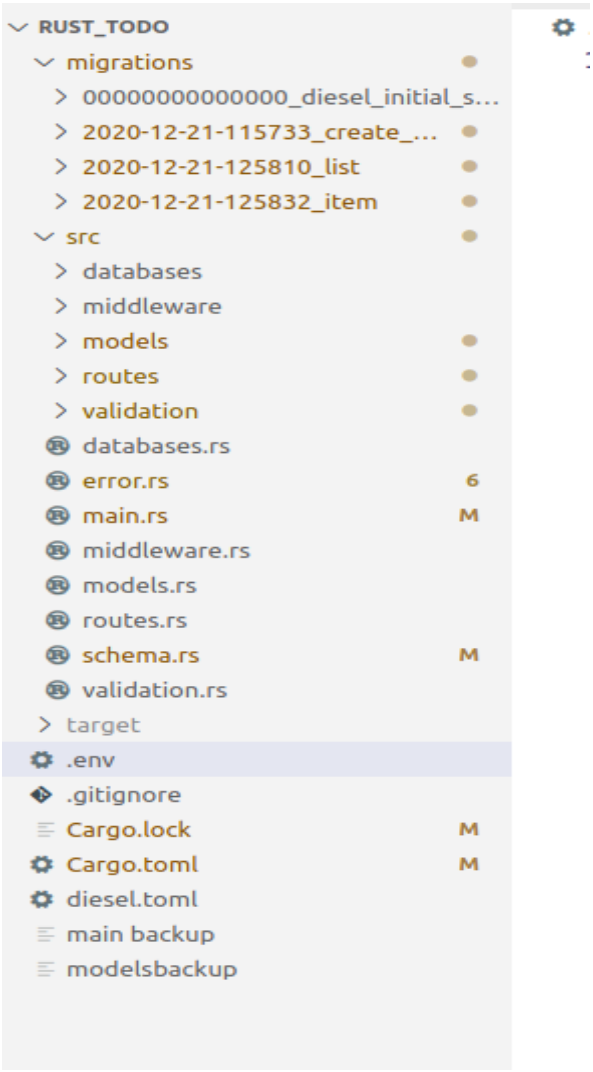
Potpis je dio tokena koji se stvara pomoću kodiranog zaglavlja i nosivosti, tajnog ključa i algoritma koji je naveden u zaglavlju. Ovaj potpis koristi se kako bi se verificiralo da se poruka i njen sadržaj nisu izmijenili na putu do odredišta, a ako se koriste privatni i javni ključ za kriptiranje podataka može se provjeriti i je li poruka došla od očekivanog pošiljatelja. Nakon generiranja svih dijelova token-a oni se slažu jedan za drugim u obliku xxxx.yyyy.zzzz gdje je xxxx kodirano zaglavlje, yyyy kodiran sadržaj i zzzz potpis.

4. APLIKACIJE TODO LISTE NAPISANA U RUST-U

Jedini uvjet za ispravan rad ove aplikacije je pokretanje Docker slike s PostgreSQL bazom, postavljanje točne adrese za konekciju na bazu u „*.env*“ datoteku aplikacije i pokretanje migracija kako bi se kreirali svi modeli u bazi koji su potrebni za rad same aplikacije. Pošto aplikacije nema „*front-end*“ poželjno je imati alat kako bi se mogli slati zahtjevi na bazu i primati odgovori na te zahtjeve, u ovom projektu korišten je alat Postman, te je isto dobro imati alat s kojim se možemo spojiti na bazu i dobiti grafički prikaz sadržaja te baze, u ovom projektu za to korišten je DBeaver.

4.1. Struktura aplikacije

Slika 4.1 prikazuje samo strukturu aplikacije, tj. način na koji su određeni dijelovi kôda podijeljeni u svoje zasebne logičke cjeline. Razlog ovakve strukture projekta je vođenje idejom da se kôd treba odvajati u cjeline gdje će svaka cjelina biti odgovorna za samo jedan dio funkcionalnosti aplikacije.



Slika 4.1. *prikaz strukture aplikacije.*

Pod „*databases*“ se nalazi kôd za kreiranje tzv „*pooled*“ konekcije na bazu,pokazan na slici 4.2, pošto je broj konekcija ograničen, a stvaranje nove konekcije prilikom svakog novog zahtjeva nije efikasno. Kreira se udružena veza (engl. *Pooled connection*), koja se kopira i prosljeđuje u svaki servis po potrebi. Iz okruženja aplikacije se čita „DATABASE_URL“ te se pokušava uspostaviti poveznica na bazu, a u slučaju neuspjeha povezivanja dobiva se greška s porukom „*Database not found*“ i aplikacija se neće pokrenuti.

```rust
pub fn establish_connection() -> Pool {
    dotenv::dotenv().ok();
    let database_url = std::env::var("DATABASE_URL").expect("Database not foudn");
    Pool::builder()
        .build(ConnectionManager::<PgConnection>::new(database_url))
        .unwrap()
}
```

Slika 4.2. *Prikaz kôda povezivanja na bazu.*

Pod „*middleware*“, na slikama 4.3 i 4.4, se nalazi posrednički softver (engl. *Middleware*). To su funkcije koje se pokreću prije nego što bilo koji zahtjev dođe do uslužnog sloja (engl. *Service layer*) aplikacije kako bi se ti zahtjevi provjerili i potvrdili.

Na slici 4.3 vidi se dio za ovjeru zahtjeva. Prvo se pokušava pronaći „*Authentication*“ zaglavlje (engl. *Header*), u slučaju da se to ne uspije dobit ćemo grešku s porukom „ *no auth header found*“. Ako je zaglavlje pronađeno pokušat će se raščlaniti ga (engl. *Parse*) u tekstualni format i iz njega iščitati tzv. „*jwt*“ (engl. *Json web token*). Ako se iz zaglavlja uspješno iščitao „*jwt“* on se dodaje u proširenja (engl. *extenstions*) od HTTP zahtjeva, gdje će se kasnije pomoću tih proširenja provjeriti je li taj HTTP zahtjev autoriziran.

```
fn call(&mut self, req: ServiceRequest) -> Self::Future {
    if req.path() == "/login" || req.path() == "/register" {
        return Either::Left(self.service.call(req));
    };
    let jwt = match req.headers().get("Authorization") {
        Some(header) => match header.to_str().ok() {
            Some(val) => val.to_string(),
            None => {
                return Either::Right(ok(req.into_response(
                    HttpResponse::Unauthorized()
                        .json(json!({
                            "message":"Could not parse header"
                        }))
                        //.header(http::header::LOCATION, "/login")
                        //.finish()
                        .into_body(),
                )));
            }
        },
        None => {
            return Either::Right(ok(req.into_response(
                HttpResponse::Unauthorized()
                    .json(json!({
                        "message":"no auth header found"
                    }))
                    //.header(http::header::LOCATION, "/login")
                    //.finish()
                    .into_body(),
            )));
        }
    };
    match verify(jwt) {
```

Slika 4.3. *Prvi dio posredničkog softvera za ovjeru (engl. Authentication).*

Na slici 4.4 je kôd za provjeru iščitanog jwt-a da se provjeri je li zahtjev poslan od ovjerenog korisnika, ako nije dobit će se greška s porukom „*Could not verify json*“. Ako je traženo zaglavlje pronađeno i uspješno ovjereno zahtjev se nastavlja dalje obrađivati. Ova ovjera zahtjeva se radi na svaki zahtjev osim ako je on poslan na krajnju točku (engl. *Endpoint*) „*/login*“ ili „*/register*“ jer na tim krajnjim točkama korisnik ne treba biti prijavljen.

```
};
match verify(jwt) {
    Ok(user) => {
        req.extensions_mut().insert(user);
        return Either::Left(self.service.call(req));
    }
    Err(e) => {
        println!("{}", e);
        return Either::Right(ok(req.into_response(
            HttpResponse::Unauthorized()
                .json(json!({
                    "message":"Could not verify json"
                }))
                //.header(http::header::LOCATION, "/login")
                //.finish()
                .into_body(),
        )));
    }
}
```

Slika 4.4. *Drugi dio posredničkog softvera za ovjeru.*

Na slici 4.5. vidi se dio kôda za generiranje i provjeru „*jwt*“-a. U ovom slučaju u „*Json web token*“ sprema se cijeli korisnikov objekt, koji sadrži „*user_id*“, „*pword*“ i „*username*“ ključeve zajedno s njihovim vrijednostima, te „*exp*“ koji govori koliko dugo je jedan „*jwt*“ valjan, u ovom slučaju to je 10 minuta. Ključ koji sadržava korisnikovu kriptiranu zaporku se inače ne bi trebao spremati u „*jwt*“ pošto ako ga se presretne može se vidjeti njegov sadržaj, ali za primjene ovog programa sprema se kako bi se pokazalo da se zaporke ne spremaju nigdje kao običan tekst.

„*Jwt*“ generira se pomoću tajnog enkripcijskog ključa, u ovom slučaju „*MyJwtSecret*“, koji je poznat samo samoj aplikaciji, te se isti taj ključ koristi kako bi se verificirao bilo koji dobiveni „*jwt*“.Kako bi se bilo koji zahtjev prema serveru smatrao valjanim, on mora sadržavati „*jwt*“ koji je potpisan s pravim ključem, te nije istekao. Za enkripciju koristi se SH256 algoritam.

```rust
pub fn generate(user: User) -> String {
    let secret: String = String::from("MyJwtSecret");
    let duration: i64 = 1200;
    let exp: DateTime<Utc> = Utc::now() + chrono::Duration::seconds(duration);

    let claims: Claims = Claims {
        user,
        exp: exp.timestamp(),
    };

    jsonwebtoken::encode(
        header: &jsonwebtoken::Header::default(),
        &claims,
        key: &jsonwebtoken::EncodingKey::from_secret(&secret.as_bytes()),
    ): Result<String, Error>
    .unwrap_or_default()
}

pub fn verify(token: String) -> Result<User, jsonwebtoken::errors::Error> {
    let secret: String = String::from("MyJwtSecret");

    let data: TokenData<Claims> = jsonwebtoken::decode::<Claims>(
        &token,
        key: &jsonwebtoken::DecodingKey::from_secret(secret.as_bytes()),
        validation: &jsonwebtoken::Validation::new(alg: jsonwebtoken::Algorithm::HS256),
    )?;

    Ok(data.claims.user)
}
```

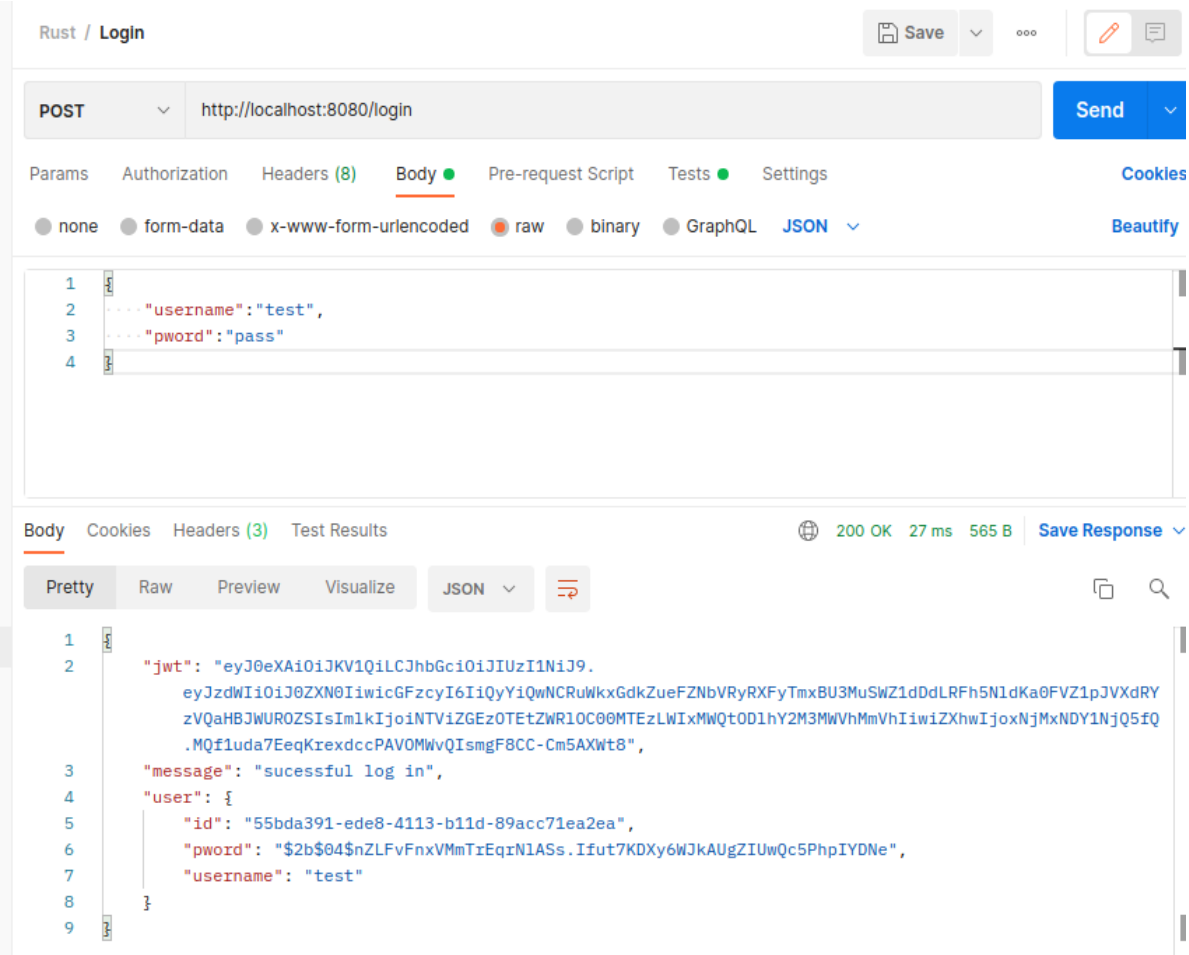
Slika 4.5. *Slika s funkcijama za generiranje i provjeru jwt-a*

Pod „*validation*“, na slici 4.6., nalazi se dio kôd-a za validaciju sadržaja zahtjeva. Svaki zahtjev u ovoj aplikaciji je JSON objekt, koji se sadrži od ključeva i pripadajućim vrijednostima. Ova funkcija prima JSON objekt, i pravila po kojima se taj objekt treba ovjeriti. Provjerava se postoje li ključevi koji su u pravili definirani kao potrebni (engl. *Required*) i jesu li ti ključevi odgovarajući tip podataka.

```rust
pub fn validate<T>(item: web::Json<Value>, keys: Vec<&str>) -> Result<T, Error>
where
    T: Clone + for<'de> Deserialize<'de> + Debug,
{
    let mut messages: Vec<Value> = vec![];
    for key in keys {
        let rule = String::from(key);
        let mut rules: Vec<&str> = rule.split('|').collect();
        let value_key = rules.remove(0);
        for rule in &rules {
            if item[value_key] == Value::Null {
                messages.push(json!({
                    "message": format!("{} not found in body", value_key)
                }));
            } else {
                match checktype(&item[value_key], rule) {
                    Err(e) => {
                        messages.push(json!({
                            "message":
                                format!(
                                    "{} is not of required type {}, it is {}",
                                    value_key, rule, e
                                )
                        }));
                    }
                    _ => {}
                }
            }
        }
    }
    if messages.len() == 0 {
        let stuff = to_string(&item.into_inner())
            .map_err(|e| Error::from(e).to_response())
            .unwrap();
        let data: T = serde_json::from_str(&stuff)
            .map_err(|err| Error::from(err).to_response())
            .unwrap();
        return Ok(data);
    } else {
        return Err(Error::from(json!({ "messages": messages })));
    }
}
```

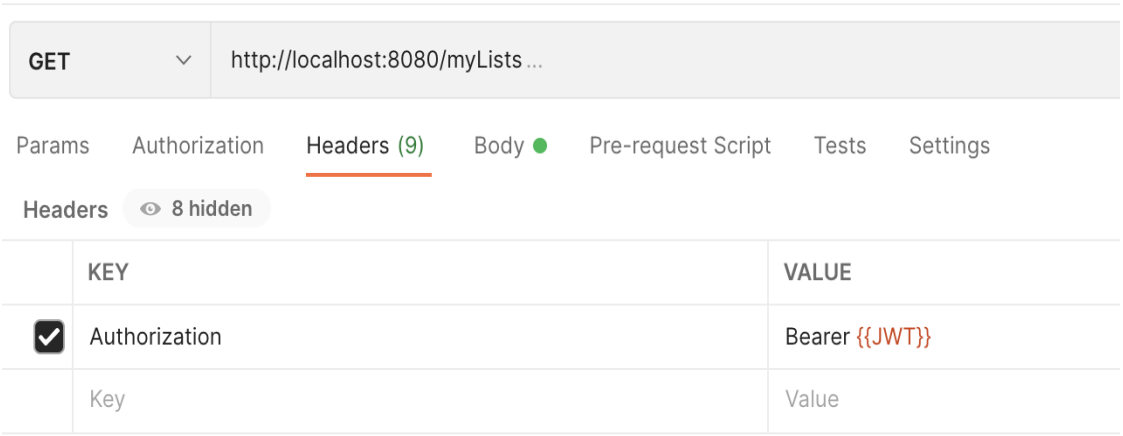
Slika 4.6. *Kôd* za provjeru sadržaja zahtjeva.

Na slici 4.7. vidi se uspješan zahtjev za prijavu u aplikaciju. Zahtjev se sadrži od objekta koji se sastoji od dva kljuca „*username*“ s vrijednošću „test“ i „*pword*“ s vrijednošću „*pass*“. Na donjem dijelu slike vidi se i odgovor iz naše aplikacije. U odgovoru se nalazi „*jwt*“, poruka da je prijava uspješna, i objekt korisnika koji se prijavio s kriptiranom zaporkom. Nakon uspješne prijave ovaj korisnim mora u svakom svom zahtjevu slati „*jwt*“.



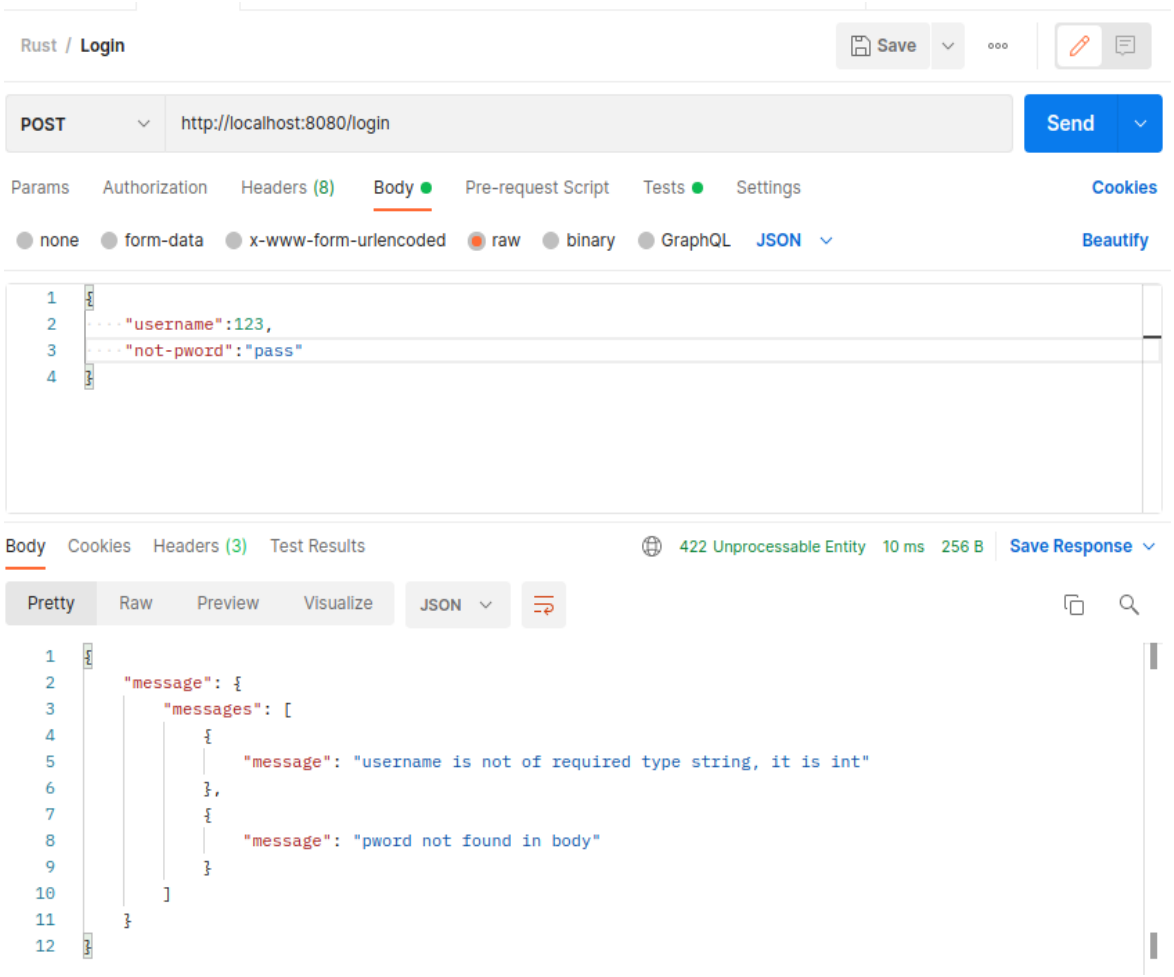
Slika 4.7. *Primjer uspješnog zahtjeva za priajvu iz alata Postman.*

Na slici 4.8. vidi se kako se nakon uspješne prijave prosljeđuje „*jwt*“ token u zaglavlje svakog sljedećeg HTTP zahtjeva. Na slici sadržaj „*{{JWT}}*“-a je isti onaj pod ključem „*jwt*“ sa slike 4.7.



Slika 4.8. *Primjer prosljeđivanja autorizacijskog zaglavlja koji sadrži „jwt“.*

Na slici 4.9. se vidi neuspjeli zahtjev sa status kôdom 422 „*Unprocessable entity*“ s odgovarajućim porukama. Vrijednost za ključ „*username*“ treba biti tekstualnog tipa, a traženi ključ „*pword*“ uopće ne postoji u zahtjevu.



Slika 4.9. *Primjer neuspjelog zahtjeva za prijavu zbog neispravnog sadržaja zahtjeva.*

Na slici 4.10. se nalazi kôd za prijavu korisnika. Prvo se radi validacija sadržaja zahtjeva, vidi se da se u validacijsku funkciju „json_validation“ prosljeđuje „newuser“ objekt koji je poslan iz Postman-a te pravila za validaciju u formatu niza *string*-ova, gdje se svaki od njih sastoji od ključa koji se provjerava i tipom podataka za provjeru tipa tog ključa ako postoji u zahtjevu. Nakon uspješne validacije zahtjeva, pokušava se prijaviti korisnik s poslanim podatcima, ako su podatci u redu generira se „jwt“ za korisnika i vraća se nazad kao odgovor na zahtjev, ako podatci nisu točni vraća se greška s porukom „*incorrect credentials*“.

```rust
pub async fn login(conn: web::Data<Pool>, newuser: web::Json<Value>) -> impl Responder {
    let data: UserNew = match json_validation::validate::<UserNew>(
        newuser,
        vec!["username|string", "pword|string"],
    ) {
        Err(err) => return Err(err.to_response()),
        Ok(data) => data,
    };
    User::login(
        &conn.get().unwrap(),
        data.username.to_string(),
        data.pword.to_string(),
    )
    .map(|mut user| {
        if user.len() > 0 {
            let jwt = generate(user[0].clone());
            HttpResponse::Ok().json(json!({
                "message":"sucessful log in",
                "user":user.pop(),
                "jwt":jwt
            }))
        } else {
            HttpResponse::Ok().json(json!({
                "message":"incorrect credentials"
            }))
        }
    })
    .map_err(|err| err.to_response())
}
```

Slika 4.10. *Primjer k*ôda za prijavu korisnika.

U „main.rs“ file-u,prikazan na slici 4.11. nalazi se kôd za pokretanje servera na određenoj adresi i portu, zajedno s registracijom ruta kako bi im se moglo pristupiti kroz Postman, te s omotavanjem (engl. *Wrapping*) posredničkog softvera za prijavu i validaciju zahtjeva.

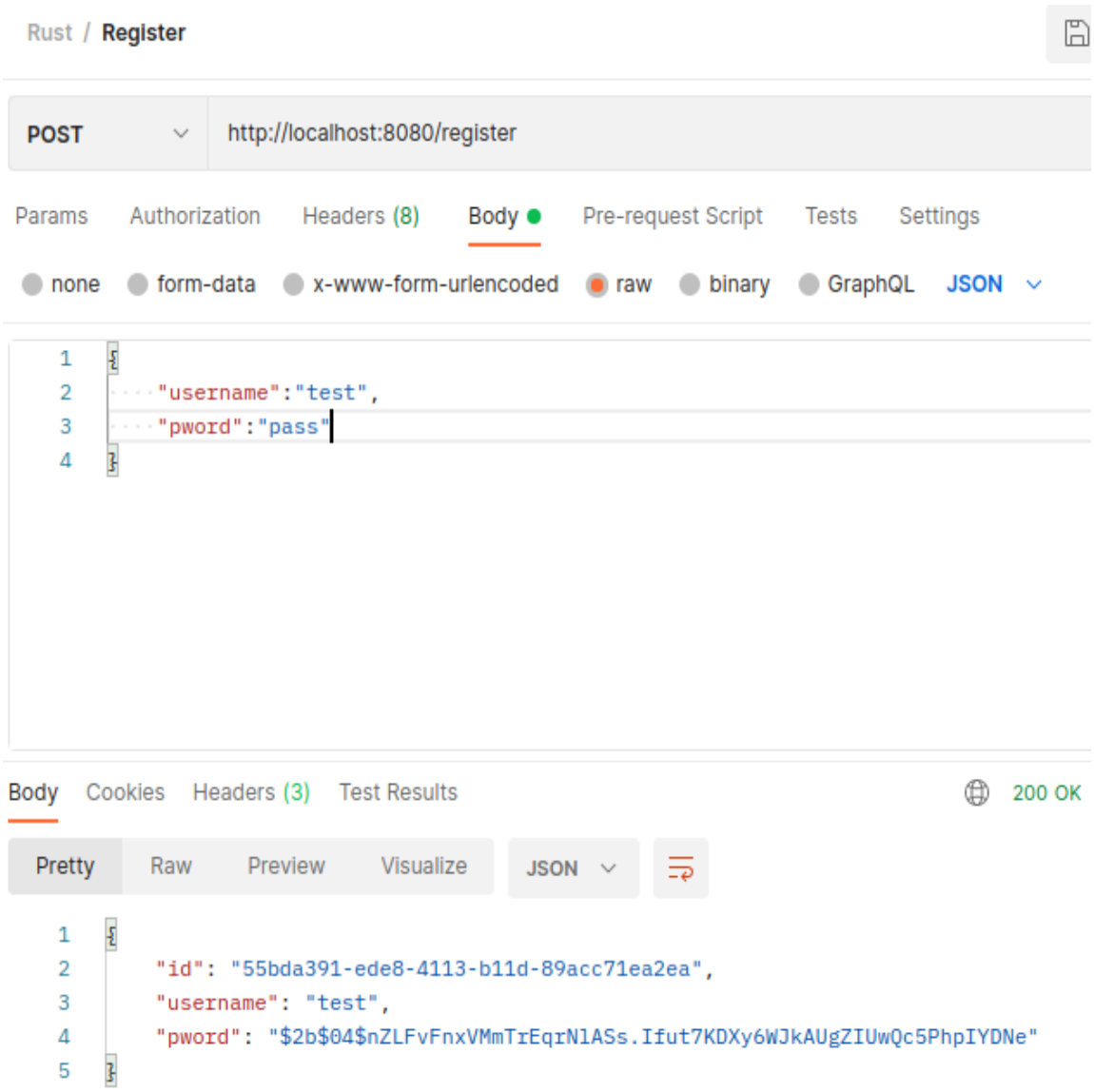
Prvo se stvara udružena veza (engl. *pooled connection*) na bazu, te se nakon kreiranja servera ta veza predaje serveru. Nakon kreiranja servera lančano se dodaju rute koje će aplikacija podržavati, prilikom dodavanja svake rute navedena je metoda koja je dopuštena na toj ruti, te funkcija koja će se pozvati na toj ruti. Zadnji poziv je dodavanje posredničkog softvera na sve rute u aplikaciji.

```rust
async fn main() -> std::io::Result<()> {
    let pool = db::establish_connection();

    HttpServer::new(move || {
        App::new()
            .data(pool.clone())
            .route("/lists", web::get().to(todolist_routes::lists))
            .route("/addlist", web::post().to(todolist_routes::addlist))
            .route(
                "/listbyid/{list_id}",
                web::delete().to(todolist_routes::delete_list),
            )
            .route(
                "/listbyid/{list_id}",
                web::get().to(todolist_routes::list_by_id),
            )
            .route("/additem", web::post().to(routes::todoitem_routes::additem))
            .route(
                "/listitems/{list_id}",
                web::get().to(routes::todoitem_routes::items_from_list),
            )
            .route(
                "/deleteitem/{item_id}",
                web::delete().to(routes::todoitem_routes::delete_item),
            )
            .route(
                "/checkitem/{item_id}",
                web::get().to(routes::todoitem_routes::check_item),
            )
            .route(
                "/uncheckitem/{item_id}",
                web::get().to(routes::todoitem_routes::uncheck_item),
            )
            .route("/register", web::post().to(user_routes::create_user))
            .route("/login", web::post().to(user_routes::login))
            .route("/myLists", web::get().to(user_routes::my_lists))
            .route("/test", web::get().to(routes::todoitem_routes::return_ok))
            .wrap(authorize::CheckLogin)
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

Slika 4.11. *kôd za kreiranje servera.*

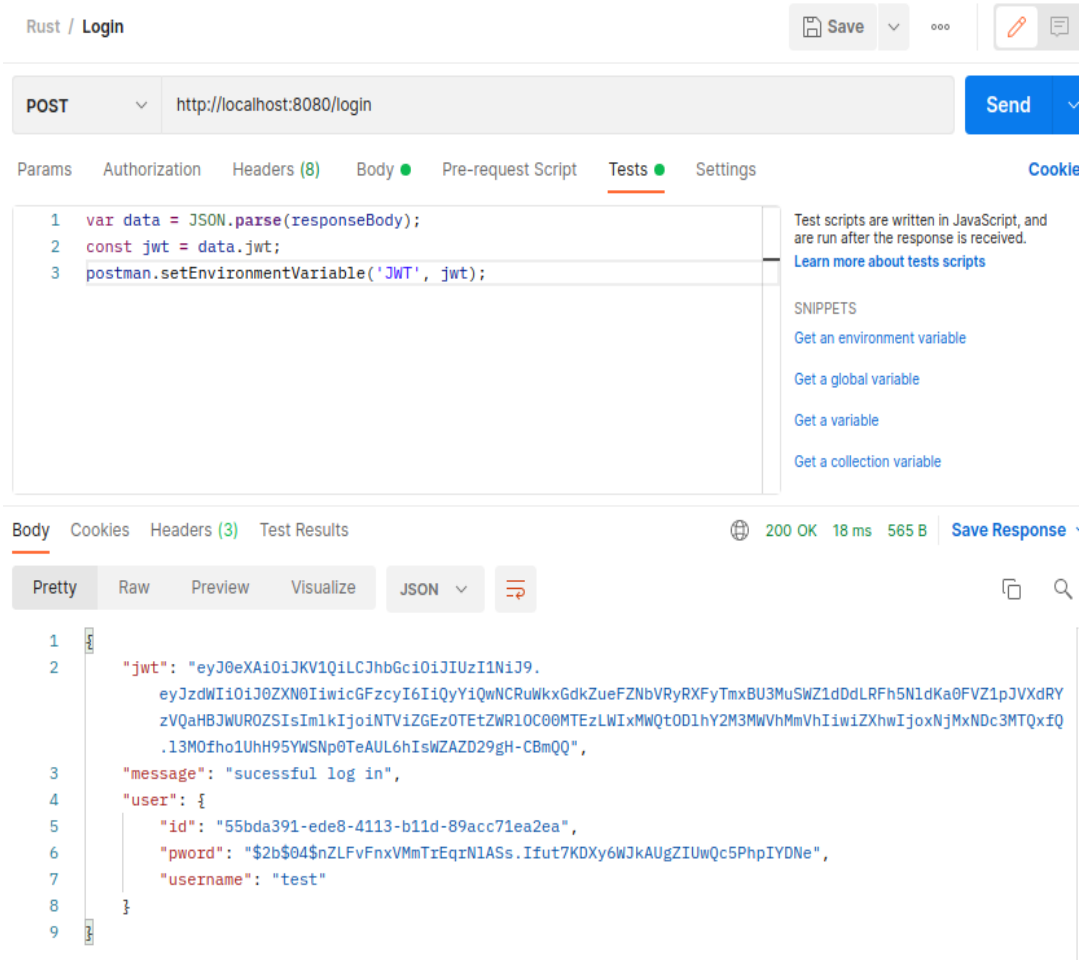
4.2. Primjer korištenja

Na slici 4.12. pokazan je primjer uspješne registracije. Zahtjev za registraciju šalje se na „*{host}/register*“, gdje „*{host}*“ može zamijeniti s bilo kojom adresom za server na kojoj je aplikacija trenutno pokrenuta, u ovom slučaju ona je pokrenuta lokalno na računalu pa je localhost. Ako je registracija uspješna kao odgovor na zahtjev dobivamo objekt kreiranog korisnika.



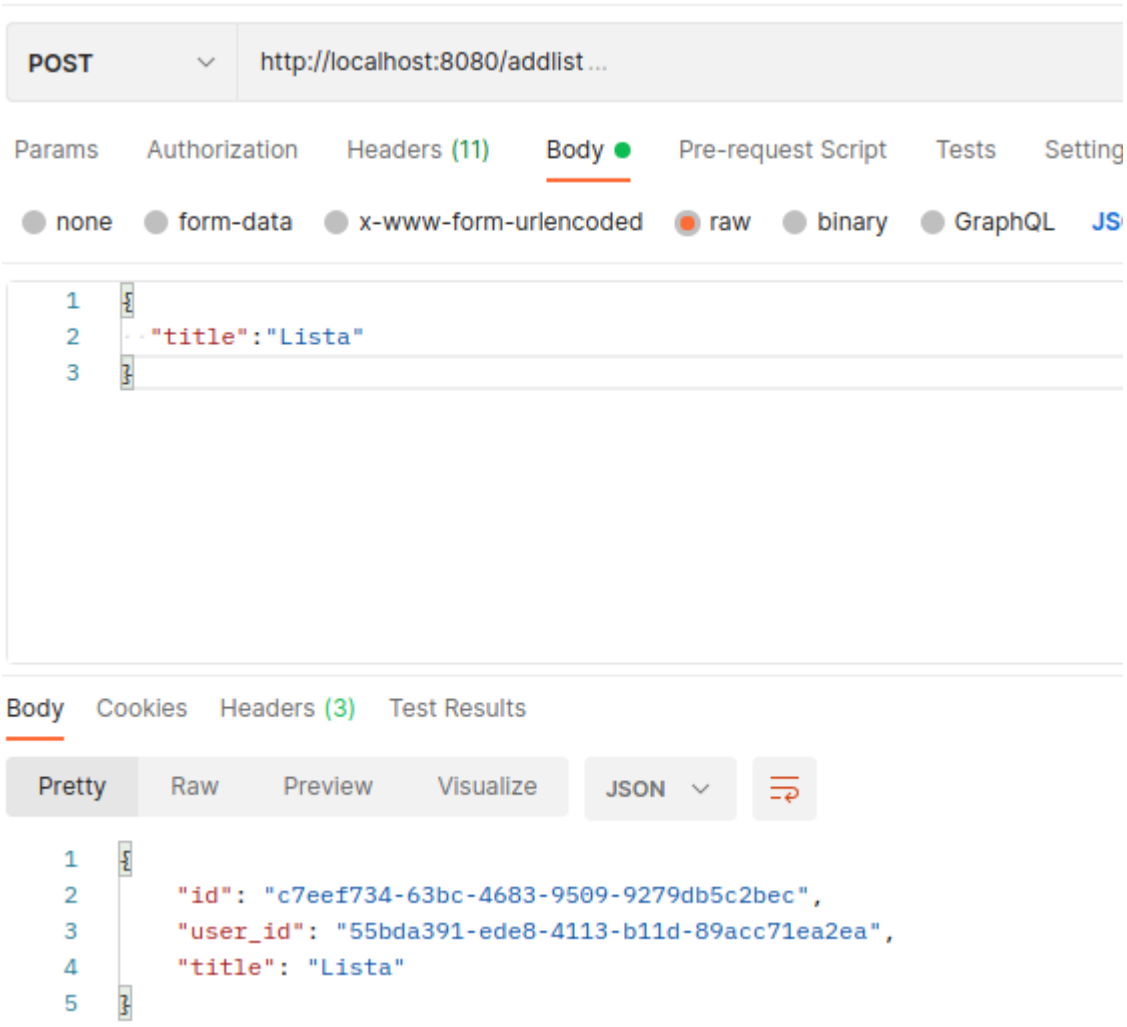
Slika 4.12. *Primjer zahtjeva za registraciju.*

Na slici 4.13. se vidi primjer uspješne prijave u aplikaciju, slično kao na slici 4.10., ali na ovoj slici umjesto „*JSON*“ sadržaja zahtjeva vidi se par linija kôda koje će se izvršiti nakon što dobijemo odgovor na zahtjev. Ova skripta će iz dobivenog odgovora spremiti „jwt“ u varijablu okruženja (engl. *Environment variable*) koja se zove „JWT“ i dodati tu varijablu u zaglavlja svih budućih zahtjeva.



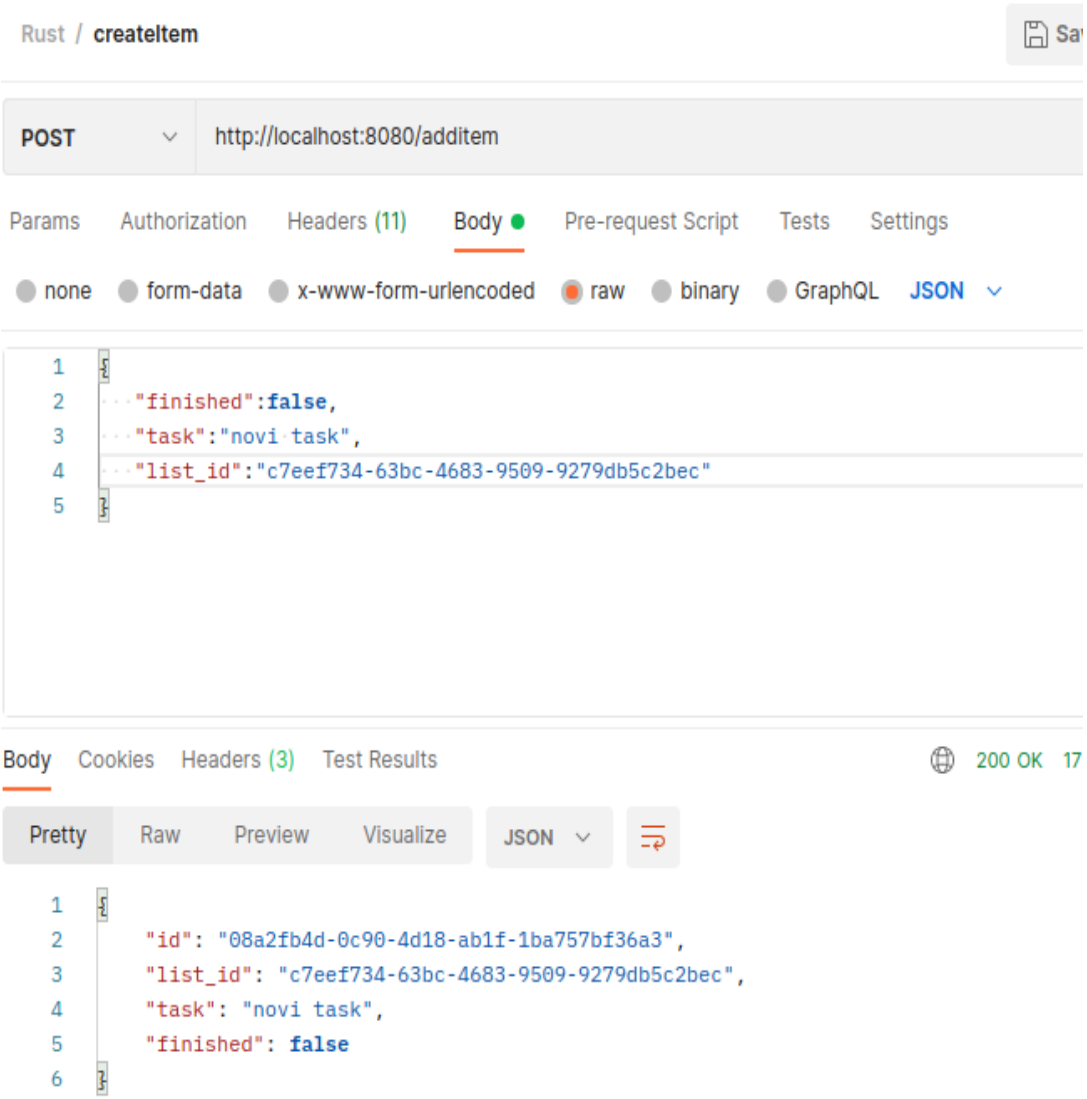
Slika 4.13. *Primjer zahtjeva za prijavu sa post request skriptom.*

Na slici 4.14. pokazan je uspješan zahtjev za kreiranje nove liste, sastoji se od ključa „*title*“ i vrijednosti tog ključa „*Lista*“, u donjem dijelu slike vidi se dobiveni odgovor iz aplikacije. U odgovoru se vidi ključ „*id*“ koji je primarni ključ za listu, i ključ „*user_id*“ koji je primarni ključ korisnika, s ovime je jednoznačno određeno da ovoj listi samo ovaj korisnik smije pristupiti.



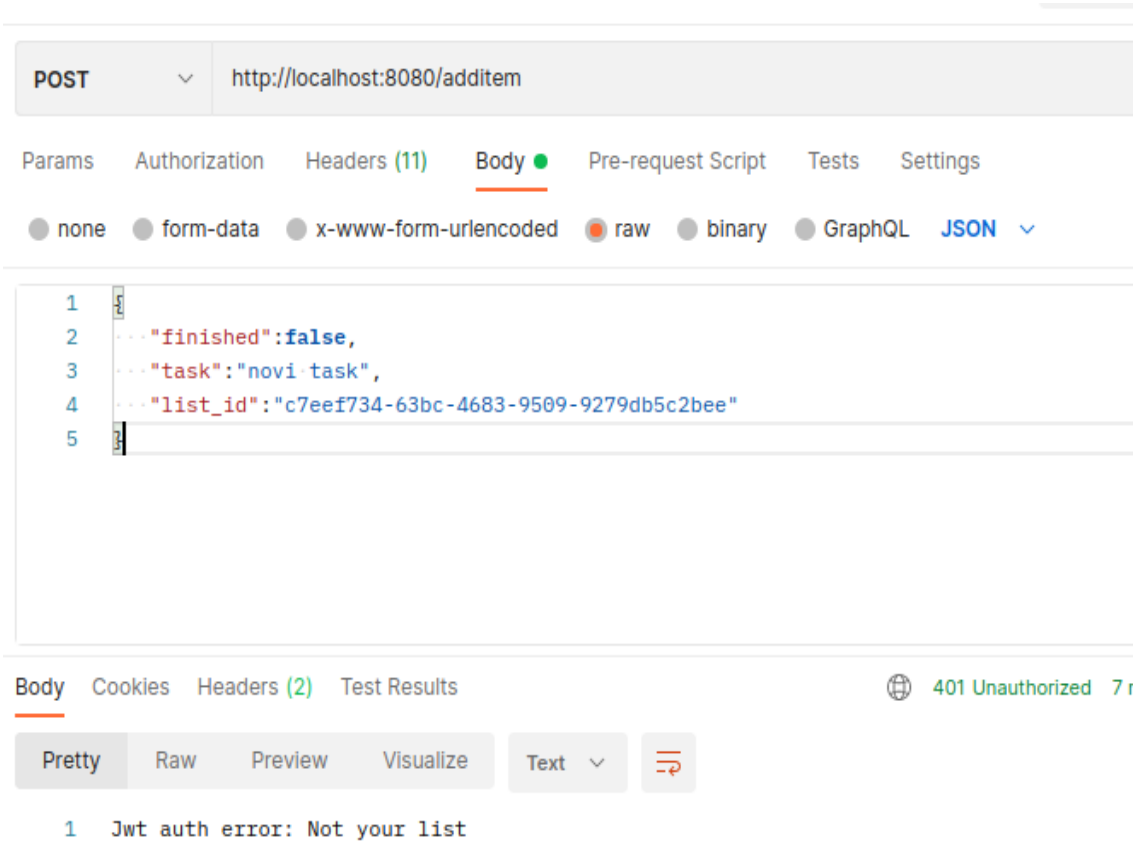
Slika 4.14. *Primjer uspješnog zahtjeva za kreiranje nove liste.*

Na slici 4.15. vidi se primjer kreiranja tj. dodavanja stavke u TODO listu, zahtjev se sastojao od ključa „*finished*“ i „*list_id*“ koji je primarni ključ neke liste, i u ovom slučaju određuje listu kojoj se ova stavka želi dodati, te ključa „*task*“ koji je naziv ove stavke u listi.



Slika 4.15. *Primjer uspješnog zahtjeva za dodavanje stavke u listu.*

Ako je korisnik pokušao pristupiti listi koja ne postoji ili nije njegova i pokušao dodati novu stavku na tu listu, dobiti će odgovor kao na slici 4.16. s http status kôdom 401 koji je govori da je zahtjev odbijen jer korisnik nema prava pristupa traženom resursu



Slika 4.16. *Primer neuspješnog zahtjeva za dodavanje stavke u listu.*

Na slici 4.17. se vidi kôd za dodavanje stavke. Prolazimo u petlji kroz sve liste korisnika, i ako nijedna nije jedna od onih u koju se stavka želi dodati, vraćamo grešku. Vidimo funkciju „*addItem*“ koja kao parametre prima konekciju, „*JSON*“ objekt koji je nova stavka koja se dodaje na listu, te cijeli HTTP zahtjev.

U tijelu funkcije „*additem“* više puta vidimo „*match*“ što je u Rust-u sličan „*if“* grananju u ostalim jezicima. Ova funkcija vraća kao rezultat bilo koji tip koji implementira osobinu (engl. *trait*) „*Responder“*, što u ovom kontekstu znači bilo koji tip podataka koji se može pretvoriti u, ili sami HTTP odgovor.

„*Match*“ poziva se na funkciji „*validate“* koja vraća rezultat koji je u Rust-u tipa „*Result<T,Error>“*, gdje je T (u Rust-u „*Ok(T)*“) bilo koji tip podataka koji očekujemo kao pravi rezultat funkcije, a *Error (* u Rust-u „*Err(Error)*“) greška koju funkcija može vratiti. U ovom slučaju nam je generički tip T nova stavka koju želimo dodati tj. „*TodoItemNew*“. U „*Ok“* grani *Match*-a u varijablu „*data*“ spremamo vrijednost te nastavljamo normalan rad funkcije „*additem“*, ako je funkcija „*validate“* rezultirala greškom, u „*Error“* grani *Match*-a ćemo vratiti

dobivenu grešku, koju pretvaramo u HTTP odgovor pozivom funkcije „*to_response“*, kao rezultat funkcije „*additem“*.

Sljedeće u tijeku funkcije vidimo dohvaćanje objekta korisnika iz zahtjeva. Na HTTP zahtjevu („*req“*) pozivamo funkciju „*extenstions_mut“* koja kao rezultat vraća promjenjivu referencu na proširenja zahtjeva, odmah nakon poziva te funkcije na dobivenim proširenjima se poziva funkcija „*remove::<User>*“ koja će iz proširenja probati izvaditi (zbog ovog vađenja je nužno da nam referenca bude promjenjiva) objekt korisnika tj. „*User“* objekt. Pošto funkcija „*remove“* kao rezultat vraća „*Option<T>“*, gdje je T u ovom slučaju tipa „*User“*, a druga varijanta „*Option“*-a je „*None“* što je u Rust-u jednako „*undefined“*, postoji mogućnost da nećemo pronaći korisnika u zahtjevu, što znači da korisnik koji je poslao zahtjev nije provjeren.

Kako bi se pokrila mogućnost da korisnik ne mora biti u svakom zahtjevu poziva se funkcija „*ok_or_else“* na rezultatu koji se dobije iz funkcije „*remove“*, Ova će funkcija za bilo kakav rezultat koji nije tipa „*User“* generirati HTTP odgovor s greškom „*User not found in request“* i vratiti taj odgovor kao rezultat funkcije „*addItem“*. Ako je rezultat funkcije „*remove“* ipak odgovarajućeg tipa, taj rezultat se sprema u varijablu „*user“*.

U nastavku toka funkcije ponovo radimo „*match“* gdje kao rezultat funkcije „*user_lists“* očekuju se sve liste korisnika koji je poslao ovaj zahtjev, kako bi se provjerilo je li lista na koju korisnik želi dodati stavku jedna od njegovih. Ovo provjeravanje se radi u „*for“* petlji.

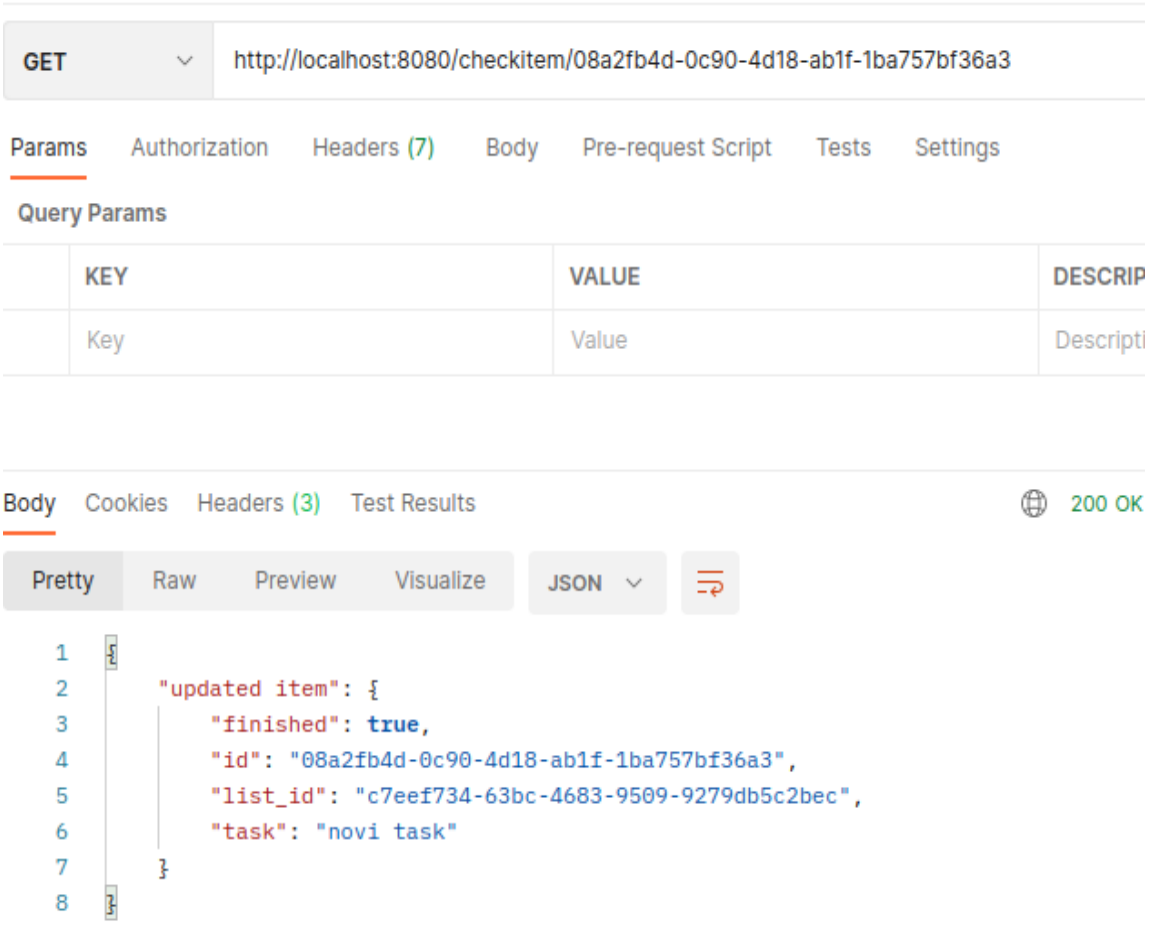
Ako se u petlji naiđe na listu gdje „*id*“ te liste odgovara „*id“*-u liste u koju se stavka želi dodati, pokušava se izvršiti upisivanje nove stavke u odgovarajuću listu pozivom funkcije „*create_item“*. Ova funkcija kao rezultat vraća „*Result<TodoItem,Error>“* tj. vratit će se novo-upisana stavka ili greška. Pošto se nakon samog poziva neke funkcije u Rust-u ne zna kojeg je tipa njen rezultat, u ovom slučaju to može biti „*Ok(TodoItem)“ ili „Err(Error)“*, pozivamo ulančano dvije funkcije „*map“* i „*map_err“*, gdje će funkcija „*map“* sadržaj u „*Ok“* dijelu rezultata pretvoriti u HTTP odgovor, a „*map_err“* sadržaj u „*Err“* dijelu rezultat pretvoriti u HTTP odgovor. U oba slučaja imamo HTTP odgovor kao rezultat te ga možemo proslijediti van funkcije „*additem“*.

Samim ovime vidi se kako u Rust-u ne postoji mogućnost da se u bilo kojem trenutku dobije neočekivani tip podatka za bilo koji podatak. U Rust-u svaka funkcija ima točno određeno kakav rezultat može vratiti, te prilikom rada s rezultatima bilo koje funkcije moramo se pobrinuti da smo spremni za rad s bilo kojim od mogućih tipova rezultata, što se u Rust-u vrlo često radi sa „*match“* grananjem.

```rust
pub async fn additem(
    conn: web::Data<Pool>,
    newitem: web::Json<Value>,
    req: web::HttpRequest,
) -> impl Responder {
    let data: TodoItemNew = match json_validation::validate::<TodoItemNew>(
        item: newitem,
        keys: vec!["task|string", "list_id|string"],
    ) {
        Err(err: Error) => return Err(err.to_response()),
        Ok(data: TodoItemNew) => data,
    };

    let user: User = req.extensions_mut().remove::<User>().ok_or_else(err: || {
        Error::throw(err: "Unauthorized", message: Some("User not found in request")).to_response()
    })?;

    match User::user_lists(conn: &conn.get().unwrap(), user) {
        Ok(lists: Vec<TodoList>) => {
            for list: TodoList in lists {
                if list.id == data.list_id {
                    return TodoItem::create_item(conn: &conn.get().unwrap(), item: data): Result<TodoItem, Err
                        .map(op: |item: TodoItem| HttpResponse::Ok().json(item)): Result<Response, Error>
                        .map_err(op: |err: Error| err.to_response());
                }
            }
            return Err(Error::throw(err: "Unauthorized", message: Some("Not your list")).to_response());
        }

        Err(err: Error) => return Err(err.to_response()),
    };
}
```

Slika 4.17. *Kôd za dodavanje stavke u listu.*

Na slici 4.18 se vidi zahtjev za označavanje stavke kao gotova. U ovom slučaju primarni ključ tj. „*id*“ stavke koju želimo označiti ne šaljemo u „*JSON*“ dijelu zahtjeva već se ono nalazi u samoj ruti vidljivo gore na slici kao {host}/checkitem/{id stavke}. U slučaju da stavka ne pripada korisniku koji šalje zahtjev, on bi bio odbijen s 401 „*Unauthorized*“ greškom.



Slika 4.18. *Primjer zahtjeva da se stavka označi kao gotova.*

5. ZAKLJUČAK

U sklopu ovog rada napravljena je web aplikacija TODO liste u Rust-u koja će omogućuje korisnicima prijavu i registraciju, kreiranje i brisanje lista, dodavanje stavki na svoje liste i označavanje tih stavki kao da su još u tijeku ili završene. Aplikacija pruža svim korisnicima sigurnost da njihovim listama ne može pristupiti nitko drugi koristeći jwt kao osnovnu validaciju i provjeru autentičnosti korisnika.

Sav kôd je napisan u Rust-u osim Sql dijela kôda u kojem su migracije za kreiranje modela u bazi. Rust je jezik niske razine namijenjen za programiranje raznih sustava, što je značilo da se ovakva aplikacija mogla napisati i jednostavnije u nekom drugom programskom jeziku. Za pisanje ove aplikacije korišten je Rust okvir (engl. *Framework*) naziva „Actix“. Actix je jedan od poznatijih okvira za razvoj web aplikacija u Rust-u i izabran je zato što pruža puno manje gotovih riješenja za probleme kao što su integracija baze podataka u aplikaciju, kreiranje posredničkog softvera, validacija i autentifikacija. S time što se to sve ručno razvija i implementira u aplikaciju postiže se to da je aplikacija lakša za održavati i raditi promjene.

Ova aplikacija je samo primjer web aplikacije u Rust-u, ali uz dodatan razvoj mogla bi biti korisna svima koji imaju potrebu za nečim gdje mogu pratiti svoje „zadatke“, te bi imali pristup tome svugdje, jer jedina stvar koja bi bila potrebna da se pristupi ovoj aplikaciji je web preglednik.

LITERATURA

[1] Web sustav za upravljanje projektima, https://repozitorij.etfos.hr/islandora/object/etfos%3A245 15.09.2021

[2] Web aplikacija za objavu oglasa, https://repozitorij.etfos.hr/islandora/object/etfos%3A2791 15.09.2021

[3] Youtube, https://www.youtube.com/ 10.3.2021

[4] Rust programming language, https://www.barrage.net/blog/technology/rust-programming-language 11.09.2021.

[5] Rust book, https://doc.rust-lang.org/book/ 11.09.2021.

[6] Actix framework, https://actix.rs/docs/whatis/ 11.09.2021

[7] Postman, https://www.digitalcrafts.com/blog/student-blog-what-postman-and-why-use-it 11.09.2021.

[8] Docker, https://www.docker.com/resources/what-container 11.09.2021.

[9] Dbeaver, https://dbeaver.io/ 17.09.2021.

[10] What is PostgreSQL, https://www.postgresqltutorial.com/what-is-postgresql/ 12.09.2021.

[11] Json Web tokens, https://jwt.io/introduction 27.09.2021

SAŽETAK

Naslov: Web aplikacija TODO liste implementirane pomoću Rust programskog jezika

U sklopu završnog rada napravljena je web aplikacija koja omogućuje kreiranje TODO liste te dodavanje stavki na liste. Aplikacija omogućuje jednostavnu registraciju i prijavu, te nakon prijave je moguće kreirati više listi. Moguće je pregledati sve liste, izmijeniti ih dodavanjem ili brisanjem stavki, te je moguće svaki stavku pojedinačno označiti kao završenu. Aplikacija je napisana u Rust jeziku, koristeći Actix okvir za izgradnju web aplikacija. Kao baza podataka korišten je PostgreSQL koji je pokrenut u Docker-u. Za pristup bazi podataka korišten je DBeaver kao alat koji pruža grafičko sučelje za prikaz podataka. Aplikacija nema „front end“ tako da se za interakciju sa aplikacijom koristio alat Postman. Postman omogućuje jednostavno slanje zahtjeva prema serveru i prikaz dobivenih odgovora. Prilikom slanja svakog zahtjeva provjerava se sadržaj tog zahtjeva te se provjerava i autentičnost korisnika koji šalje zahtjev, uz to radi se i provjera pristupa resursima kako nitko ne bi uspio pristupiti tuđim listama.

Ključne riječi: baza podataka, Docker, Postman, Rust, web aplikacija

ABSTRACT

Title: TODO lists web application implemented using Rust programming language

As part of the final thesis, a web application was created that allows you to create a TODO list and add items to lists. The application allows easy registration and registration, and after logging in it is possible to create multiple lists. You can view all lists, modify them by adding or deleting items, and each item can be marked individually as completed. The application is written in Rust language, using the actix framework to build web applications. PostgreSQL was used as a database, which was launched in Docker. To access the database, DBeaver was used as a tool that provides a graphical interface for displaying data. The application does not have a "front end" so the Postman tool was used to interact with the application. Postman allows you to easily send requests to the server and view the responses received. When submitting each request, the content of that request is validated and the authenticity of the user sending the request is also verified, and resource access is checked so that no one can access other people's lists.

Keywords database, Docker, Postman, Rust, web application

ŽIVOTOPIS

Dominik Fačko, rođen je 10. prosinca 1996. godine u Osijeku. Živi u Kopačevu. Išao je u osnovnu školu u Prosvjetno kulturni centar Mađara u Republici Hrvatskoj. A srednju školu je išao je u 1. Gimnaziju Osijek. Nakon srednje škole upisuje Mathos gdje nakon dvije godine studiranja odlučuje upisati Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, smjer Informatika. Praksu je odradio u Mono-u a nakon prakse se zaposlio kao student u Barrage-u gdje i trenutno razvija svoje znanje o backend web developmentu.

PRILOZI

Projektna mapa s izvornim kôdom te priloženim word i pdf dokumentima nalazi se na optičkom disku koji je priložen uz ispisanu verziju završnog rada.