

Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli



Fakultet informatike u Puli

(7) Autentifikacija i autorizacija zahtjeva

#7

WA

Autentifikacija i autorizacija su ključni koncepti području sigurnosti informacijskih sustava, a predstavljaju srodne, ali različite procese. Autentifikacija je proces provjere identiteta korisnika ili sustava, a za cilj ima utvrditi je li osoba ili sustav onaj za koga se predstavlja. Autorizacija je proces određivanja prava pristupa korisnika ili sustava određenim resursima ili funkcionalnostima. U ovom poglavlju, naučit ćemo kako implementirati ove dva ključna procesa svake web aplikacije. Konkretno, naučit ćemo kako autentificirati korisnika na poslužitelju te poslati autorizacijski token (JSON Web Token) klijentu.

Posljednje ažurirano: 19.1.2026.

Sadržaj

- [Web aplikacije \(WA\)](#)
- [\(7\) Autentifikacija i autorizacija zahtjeva](#)
 - [Sadržaj](#)
- [1. Autentifikacija vs Autorizacija](#)
 - [1.1 Autentifikacija korisnika](#)
 - [1.2 Enkripcija vs Hashiranje](#)
 - [1.3 `bcrypt` paket](#)
 - [1.4 Registracija korisnika](#)
 - [1.5 Provjera podudaranja *hash* vrijednosti \(autentifikacija\)](#)
- [2. Autorizacija kroz JWT token](#)
 - [2.1 Što je ustvari token?](#)
 - [2.2 Kako iskoristiti JWT token za autorizaciju?](#)
 - [2.3 Provjera valjanosti JWT tokena](#)
 - [2.4 Implementacija funkcija za generiranje i provjeru JWT tokena](#)

- [1. Korak \(Registracija korisnika\)](#)
- [2. Korak \(Prijava korisnika s klijentske strane\)](#)
- [3. Korak \(Prijava korisnika na poslužiteljskoj strani\)](#)
- [4. Korak \(Generiranje JWT tokena\)](#)
- [5. Korak \(Pohrana JWT tokena na klijentskoj strani i slanje na poslužitelj\)](#)
- [6. Korak \(Provjera valjanosti JWT tokena na poslužiteljskoj strani\)](#)
- [2.5 Autorizacijski middleware](#)
- [2.6 Rok trajanja JWT tokena](#)
- [Samostalni zadatak za Vježbu 7](#)

1. Autentifikacija vs Autorizacija

Autentifikacija (eng. *authentication*) je proces provjere identiteta korisnika ili sustava. Cilj autentifikacije je utvrditi je li osoba ili sustav onaj za koga se predstavlja.

Kako funkcioniра autentifikacija u web aplikacijama?

- Prikupljanje vjerodajnica (eng. credentials):** Korisnik unosi vjerodajnice (npr. korisničko ime, lozinku, biometrijske podatke, PIN, itd.) putem određenog sučelja (npr. web obrasca).
- Provjera vjerodajnica:** Poslužitelj provjerava unesene podatke uspoređujući ih s onima pohranjenima u bazi podataka (npr. podudaranje korisničkog imena i lozinke).
- Rezultat provjere:** Ako su uneseni podaci ispravni, korisnik je autentificiran, i dobiva **autorizacijski pristup**.

Autorizacija (eng. *authorization*) je proces određivanja prava pristupa **autentificiranog** korisnika. Dakle, autorizacija dolazi nakon uspješne autentifikacije.

Kako funkcioniра autorizacija u web aplikacijama?

- Dodjela prava pristupa:** Poslužitelj upravlja pravima pristupa korisnika (npr. u obliku definiranih uloga, dozvola, itd.).
- Provjera prava pristupa:** Kada korisnik pokuša pristupiti određenom resursu ili funkcionalnosti, poslužitelj provjerava je **li taj korisnik ovlašten** za taj pristup.
- Rezultat autorizacije:** Ako je korisnik ovlašten, poslužitelj mu omogućuje pristup traženom resursu ili funkcionalnosti, inače mu vraća grešku.

Sljedeća tablica ukratko objašnjava razliku između autentifikacije i autorizacije:

Autentifikacija	Autorizacija
Provjerava tko je korisnik .	Određuje što korisnik smije raditi .
Izvodi se na početku (prijava).	Izvodi se svaki put kad korisnik traži pristup resursu.
Rezultat: "Jesi li ti stvarno ta osoba?"	Rezultat: "Smiješ li ovo raditi?"

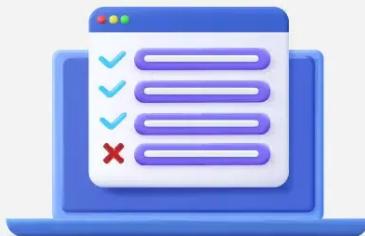
Working of Authentication and Authorization

Authentication



Confirms users are who they say they are.

Authorization



Gives users permission to access a resource

Slika 1: Razlika između autentifikacije korisnika i autorizacije zahtjeva korisnika.

U prvom dijelu ove skripte, naučit ćemo kako implementirati autentifikaciju korisnika putem jednostavnih vjerodajnica (korisničko ime i lozinka) na Express.js poslužitelju.

U drugom dijelu, naučimo kako implementirati autorizaciju putem **JSON Web Tokena (JWT)**.

1.1 Autentifikacija korisnika

Krenimo s definicijom osnovnog Express.js poslužitelja:

```
import express from 'express';
import cors from 'cors';

const app = express();
app.use(express.json());
app.use(cors());

PORT = 3000;

app.get('/', (req, res) => {
    res.send('Spremni za autentifikaciju!');
});

app.listen(PORT, () => {
    console.log(`Poslužitelj dela na portu ${PORT}`);
});
```

Rekli smo da autentifikacija uključuje prikupljanje vjerodajnica korisnika. U web aplikacijama, u pravilu se koriste **korisničko ime/email** i **lozinka** ako se radi o tradicionalnim web aplikacijama.

Međutim, u **modernim web aplikacijama**, danas su postali uobičajeni napredniji oblici autentifikacije kao što:

- **Biometrijski podaci** (npr. otisak prsta, prepoznavanje lica, glasa; biometrija se obrađuje na razini uređaja/OS-a, a ne web aplikacije)
- **Višefaktorska autentifikacija (MFA)** (kombinira više metoda autentifikacije, npr. kroz SMS, e-mail, itd.)
- **Autentifikacija bez lozinke** (npr. slanje autentifikacijskog koda na e-mail, one-time password, push notifikacije, itd.)
- **Autentifikacija bazirana na certifikatima** (npr. SSL/TLS certifikati; puno češća u B2B rješenjima, *enterprise* i internim sustavima)
- **Single Sign-On (SSO)** autentifikacija (prijava putem sigurnih poslužitelja treće strane, npr. Google, GitHub, Facebook, Microsoft, itd.)

Naravno, postoji još mnogo drugih metoda autentifikacije, a velik broj njih se međusobno kombinira. Primjerice, passwordless rješenja često kombiniraju biometriju i kriptografske ključeve pohranjene na uređaju korisnika (npr. FIDO2/WebAuthn), čime se postiže visoka razina sigurnosti uz jednostavno korisničko iskustvo.

Većina ovih metoda autentifikacije zahtijeva dodatne biblioteke i servise, te su izvan opsega ove skripte. Međutim, za vaše projekte možete pokušati implementirati neko od navedenih rješenja koristeći dostupne pakete i servise, ako hoćete.

Mi ćemo naučiti kako implementirati *from-scratch* autentifikaciju korisnika putem **jednostavnih vjerodajnica**, kao što su korisničko ime i lozinka ili e-mail i lozinka.

Definirat ćemo POST rutu putem koje će korisnik poslati svoje vjerodajnice. Do sad smo naučili da rute nazivamo prema resursima kojima pristupamo, međutim ovdje možemo uvesti iznimku jer ne pristupamo konkretnom resursu, već provodimo proces autentifikacije. Rutu možemo nazvati `/login`:

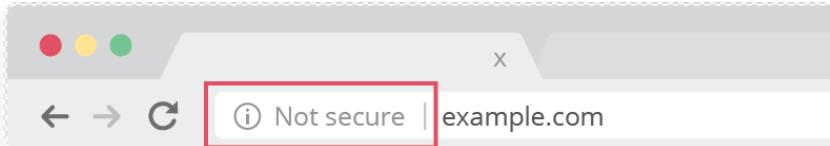
Napomena! Korisničko ime i lozinka su osjetljivi podaci te ih nikada ne želimo slati preko *route* odnosno *query* parametara, već želimo ove podatke **slati u tijelu HTTP zahtjeva!**

Napomena: Tijelo HTTP zahtjeva se smatra sigurnijim mjestom za slanje osjetljivih podataka poput lozinki jer se ne prikazuje u URL-u. Podaci poslani kroz route ili query parametre dio su URL-a, a URL se spremi u povijest preglednika, zapisuje u server logove poslužitelja (prisjetite se aplikacijskog *middlewarea*), na frontendu se prikazuje u adresnoj traci preglednika i često se prikazuje i pohranjuje u raznim alatima za nadzor i analitiku web aplikacije.

Također, slanje osjetljivih podataka nije samo po sebi sigurno ako se koristi HTTP, već postaje sigurno uz [HTTPS](#).

Buildanjem aplikacije i objavom na udaljeni poslužitelj, HTTP zahtjevi će se slati preko HTTPS (*eng. HyperText Transfer Protocol Secure*) protokola, koji osigurava siguran prijenos podataka između klijenta i poslužitelja putem enkripcije. Drugim rječima, komunikacijom preko HTTPS-a, podaci **koji se šalju između klijenta i poslužitelja su enkriptirani**, što znači da nisu čitljivi presretačima u mreži.

Web preglednici imaju ugrađene sigurnosne mehanizme koji sprječavaju slanje osjetljivih podataka preko nezaštićenih HTTP veza. Ponekad ćete dobiti upozorenje prilikom prilikom posjećivanja takvih stranica, a ponekad će vam u URL traci stajati "Not Secure" (Nije sigurno). **Čitaj:** Podaci koje unosim na takvu web stranicu/aplikaciju neće biti enkriptirani za vrijeme prijenosa preko mreže do poslužitelja, odnosno koristi se klasični HTTP protokol bez sigurnosnog sloja (TLS/SSL).



Slika 2. Primjer upozorenja u web pregledniku prilikom otvaranja non-HTTPS web stranice.

Definirat ćemo POST rutu `/login` koja će primati korisničko ime i lozinku iz **tijela zahtjeva**:

```
app.post('/login', (req, res) => {
  const { username, password } = req.body; // pristupamo korisničkom imenu i lozinci iz tijela zahtjeva
});
```

U grubo, ideja je sljedeća:

- Korisnik će poslati POST zahtjev na `/login` rutu s korisničkim imenom i lozinkom
- Poslužitelj će provjeriti jesu li korisničko ime i lozinka ispravni
- Ako jesu, korisnik je autentificiran, a poslužitelj će mu poslati potvrdu

Da ne komplikiramo, mi ćemo prvo pohraniti korisnike u *in-memory* listu korisnika:

```
const users = [
  { id: 1, username: 'johnDoe', password: 'password' },
  { id: 2, username: 'janeBB', password: 'password123' },
  { id: 3, username: 'admin', password: 'super_secret_password' }
];
```

Opisani endpoint bi izgledao ovako:

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username && user.password === password);

  if (user) {
    res.send('Uspješno ste autentificirani!');
  } else {
    res.status(401).send('Neuspješna autentifikacija!');
  }
});
```

Koje probleme ovdje možemo uočiti?

1. **Lozinke su pohranjene u običnom tekstu (eng. plain text).** Ovo je vrlo loša praksa jer ako maliciozni korisnik pristupi bazi podataka (eng. *Data leak*), može vidjeti sve lozinke korisnika. Dodatno, ako se korisnik prijavljuje preko nesigurne mreže (*non-HTTPS*), lozinka može biti presretnuta tijekom prijenosa.
2. **Nema nikakvog mehanizma zaštite od brute-force napada.** Maliciozni korisnik može pokušati beskonačno puta prijaviti se s različitim kombinacijama korisničkog imena i lozinke.

3. Pohrana lozinki bez ikakvog enkripcijskog mehanizma je **neprihvatljiva i ilegalna** u većini zemalja, posebno u zemljama Europske unije. Rizici su preveliki, a [kazne su visoke](#).
4. **Nema mehanizma za *session management***. Kako će klijent znati da je autentificiran, ako poslužitelj vrati samo poruku "Uspješno ste autentificirani!"? Bolje pitanje je: **kako će se poslužitelj sjetiti da je korisnik autentificiran**, ako je *svaki zahtjev = novi zahtjev* (HTTP je [stateless protokol](#))?

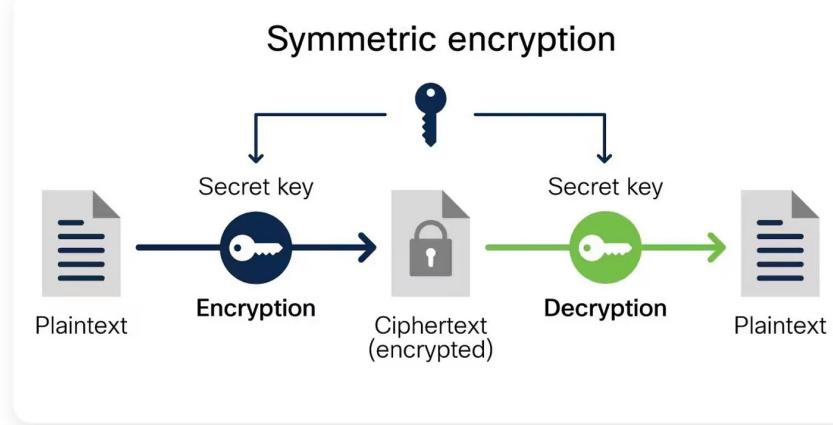
1.2 Enkripcija vs Hashiranje

Enkripcija (eng. *Encryption*) je proces pretvaranja podataka ili poruka u neki kodirani oblik kako bi se osigurala njihova privatnost i zaštita od neovlaštenog pristupa. Kodiranjem s originalni podaci, koje često zovemo "običan" ili "čisti" tekst (*plaintext*), pretvaraju u nečitki oblik koji nazivamo "šifrirani" tekst, odnosno *ciphertext*.

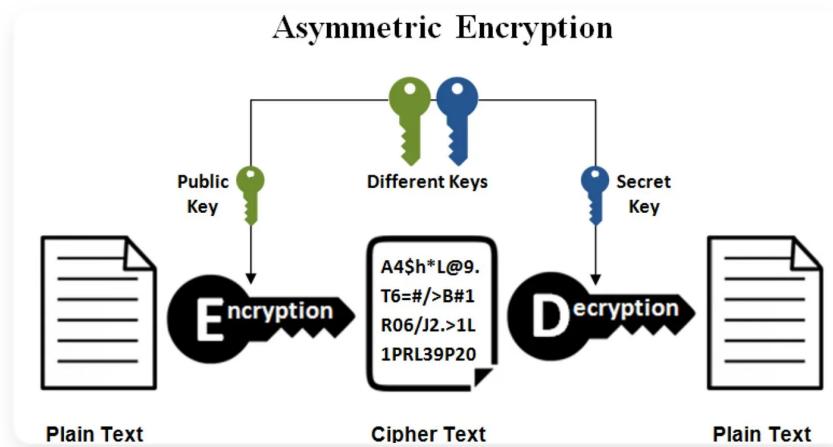
Samo osobe koje posjeduju odgovarajući **ključ** za dešifriranje mogu ponovno dobiti originalne podatke.

Postoje dvije vrste enkripcije:

1. **Simetrična enkripcija**: Koristi **isti ključ za enkripciju i dekripciju podataka**.
2. **Asimetrična enkripcija**: Koristi dva različita, ali povezana ključa: **javni ključ** za enkripciju i **privatni ključ** za dekripciju podataka. Još se naziva i *kriptografija javnog ključa* (eng. *Public Key Cryptography*).



Slika 3. **Primjer simetrične enkripcije**: koristi isti ključ za enkripciju i dekripciju podataka.

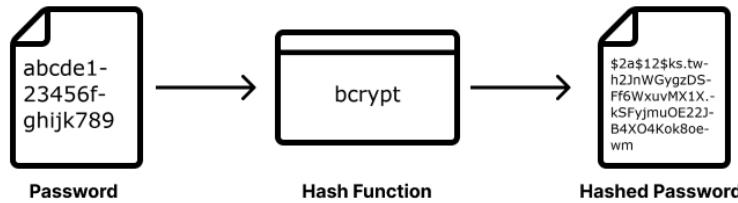


Slika 4. **Primjer asimetrične enkripcije**: koristi dva različita, ali povezana ključa: javni ključ za enkripciju i privatni ključ za dekripciju podataka.

Hashiranje (eng. *Hashing*) je proces pretvaranja ulaznih podataka u fiksni niz znakova pomoću matematičke funkcije koju nazivamo **hash funkcijom**. hash funkcija je funkcija koja prima **ulazne podatke proizvoljne duljine** i vraća **izlazni niz fiksne duljine**.

Zapamtite! Glavna razlika između enkripcije i hashiranja je da **hashiranje nije reverzibilno** (eng. *Irreversible*). To znači da ne možemo dobiti originalne podatke iz hashiranog niza ni na koji način. Hashiranje se koristi za **sigurno pohranjivanje lozinki** uz dodatne sigurnosne mehanizme kao što su dodani *salt* (nasumični niz znakova/bitova koji se *appenda* na originalnu lozinku - više u nastavku).

Password Hashing



Slika 5. **Primjer hashiranja:** ulazni podaci se pretvaraju u fiksni niz znakova pomoću tzv. *hash* funkcije.

Dakle, lozinka koja se jednom **hashira** ne može se "**dehashirati**"!

- **hash funkcije su matematički algoritmi** koji "generiraju" jedinstveni niz znakova za svaki ulazni niz, ali se iste funkcije ne mogu koristiti za "rekonstrukciju" originalnog niza
- to je zato što *hash* funkcije **gube određene informacije** prilikom generiranja *hash* vrijednosti (više različitih ulaza može imati isti *hash* – tzv. kolizije)
- *hash* funkcije **nisu bijektivne funkcije**, što znači da **nije moguće jednoznačno mapirati izlaz natrag na ulaz** - ne postoji **inverzna funkcija**

Što mislite, kako bismo onda provjerili je li korisnik unio ispravnu lozinku? 🤔

► Spoiler alert! Odgovor na pitanje

Primjer: Kako ćemo provjeriti je li unesena lozinka ispravna?

1. Pohranjeni *hash* u bazi podataka: `5f4dcc3b5aa765d61d8327deb882cf99`
2. Unesena lozinka pri prijava: `lozinka123`
3. Ponovno se izračunava *hash* unesene lozinke: `HASH(lozinka123)=5f4dcc3b5aa765d61d8327deb882cf99`
4. Usporedba *hashova*: `5f4dcc3b5aa765d61d8327deb882cf99 == 5f4dcc3b5aa765d61d8327deb882cf99`
(lozinke, odnosno **njihovi hashovi**, se podudaraju)

Prema tome, što ćemo koristiti za pohranu lozinki korisnika? **Enkripciju ili hashiranje?**

Odgovor je **hashiranje!**

Tablica u nastavku ukratko opisuje ove dvije tehnike:

Značajka	Hashiranje	Enkripcija
Smjer	Jednosmjerno – nema povratka na izvornu lozinku	Dvosmjerno – enkriptirani podatak se može dekriptirati
Svrha	Provjera lozinki bez potrebe za čuvanjem plain teksta	Čuvanje podataka u enkriptiranom obliku s mogućnošću dekripcije
Primjena	Lozinke za prijavu – uspoređuje se <i>hashirana</i> lozinka	Pohrana podataka koji se kasnije trebaju dekriptirati, primjerice u HTTPS komunikaciji
Potrebni ključevi	Nije potrebno upravljati ključevima	Potrebno je upravljati enkripcijskim ključevima
Otpornost na napade	Otpornost na brute force uz soljenje ključa	Ranjivo na kompromitaciju ključa
Upravljanje	Jednostavno – samo se <i>hash</i> funkcija i <i>salt</i> parametri pohranjuju	Puno složenije – ključevi se moraju pravilno čuvati
Tipična upotreba	Pohrana lozinki za provjeru autentičnosti	Pohrana podataka koji se trebaju dekriptirati (npr. poruke), blockchain tehnologije
Prednosti	Jednosmjernost, jednostavnost, otpornost na napade	Omogućava povrat podataka u izvornom obliku
Nedostaci	Ne omogućava povrat lozinke, samo usporedbu <i>hash</i> vrijednosti	Ako ključ procuri, svi podaci su ugroženi
Primjeri algoritama	SHA-256, bcrypt, Argon2	AES, RSA, DES

U praksi, **hashiranje** je **sigurniji** i **jednostavniji** način pohrane lozinki korisnika, dok **enkripcija ima svoje mjesto u sigurnoj komunikaciji** i **pohrani podataka** koji se trebaju dekriptirati (ali i puno drugih scenarija).

1.3 bcrypt paket

Mi ćemo koristiti **bcrypt** algoritam za *hashiranje* lozinki korisnika. Bcrypt algoritam razvili su Niels Provos i David Mazières 1999. godine, a danas je jedan od popularnijih algoritama za *hashiranje* lozinki. Koga zanima više o bcrypt algoritmu, može pročitati članak na [Wikipediji](#).

Instalirat ćemo `bcrypt` paket pomoću npm-a:

```
→ npm install bcrypt
```

Uključimo `bcrypt` u našu aplikaciju:

```
import bcrypt from 'bcrypt';
```

Za *hashiranje* lozinke koristimo metodu `hash`:

Moguće je koristiti asinkroni i sinkroni način rada s `bcrypt` paketom. Preporučuje se korištenje asinkronog načina rada jer je sigurniji i ne blokira izvođenje poslužitelja (*non-blocking*).

Metoda `hash` prima 3 argumenta:

- `plainPassword`: lozinka koju želimo hashirati u obliku običnog teksta
- `saltRounds`: broj rundi za generiranje *salt* vrijednosti
- `callback`: funkcija koja se poziva nakon što se *hashiranje* završi. Callback funkcija prima 2 argumenta: `err` i `hash`. Ako se dogodi greška, `err` će biti različit od `null`, a inače će `hash` sadržavati hashiranu lozinku.

Sintaksa:

```
bcrypt.hash(plainPassword, saltRounds, (err, hash) => {});
```

Broj rundi za generiranje *salt* vrijednosti određuje koliko će se puta izvršiti *hash* funkcija. Veći broj rundi znači da će *hashiranje* trajati dulje, ali će biti sigurnije. **Preporučuje se korištenje vrijednosti između 10 i 12.**

Primjerice, ako je `saltRounds = 3`, *hashiranje* će izgledati ovako:

1. Generira se *salt* vrijednost (nasumični niz znakova) **tri puta** kako bi se dobila konačna *salt* vrijednost (npr.
`salt = salt1 + salt2 + salt3`)
2. Nakon što je generirana konačna *salt* vrijednost, `bcrypt` koristi tu vrijednost zajedno s unesenom lozinkom kako bi generirao hash. Tijekom ovog procesa, lozinka i *salt* vrijednost prolaze kroz određeni broj iteracija (određenih sa `saltRounds` parametrom) kako bi se proizveo sigurni hash.
3. Jednom kada je *hash* generiran, poziva se callback funkcija s `hash` vrijednošću.

U callbacku možemo definirati jednostavnu obradu greške:

```

let plainPassword = 'lozinka123';
let saltRounds = 10;

bcrypt.hash(plainPassword, saltRounds, (err, hash) => {
  if (err) {
    console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
    return;
  } else {
    console.log(`Hashirana lozinka: ${hash}`);
  }
});

```

Ovaj kod će ispisati hashiranu lozinku u konzoli.

```
Hashirana lozinka: $2b$10$iyK8.vxPtPG8bArU9ucKjOF2tDqzEkFmaquk0yQRuNKKRG7/YBcyy
```

Slično kao kod rada s datotekama, osim callback pristupa možemo koristiti i Promise pristup s `bcrypt` paketom. Tada metoda `hash` vraća Promise objekt koji možemo raspakirati pomoću `then` i `catch` metoda, odnosno `async` i `await` sintakse.

```

bcrypt
  .hash(plainPassword, saltRounds)
  .then(hash => {
    console.log(`Hashirana lozinka: ${hash}`);
  })
  .catch(err => {
    console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
  });

```

Odnosno:

```

try {
  let _hash_ = await bcrypt.hash(plainPassword, saltRounds);
  console.log(`Hashirana lozinka: ${hash}`);
} catch (err) {
  console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
}

```

Sve skupa možemo zapakirati u asinkronu funkciju `hashPassword` koja će primati 2 argumenta: `plainPassword` i `saltRounds`.

```

async function hashPassword(plainPassword, saltRounds) {
  try {
    let _hash_ = await bcrypt.hash(plainPassword, saltRounds);
    return hash;
  } catch (err) {
    console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
    return null;
  }
}

```

Ova funkcija će vratiti hashiranu lozinku ako je proces uspješan, inače će vratiti `null`.

1.4 Registracija korisnika

Sada kada znamo kako hashirati lozinke korisnika, možemo implementirati funkcionalnost registracije korisnika.

Dodati ćemo rutu POST `/register` koja će primati korisničko ime i lozinku korisnika.

- dodajemo korisnika u *in-memory* listu korisnika, ali prije toga hashiramo lozinku funkcijom `hashPassword`

```
const users = [];

app.post('/register', async (req, res) => {
  const { username, password } = req.body;

  const hashed_password = await hashPassword(password, 10);

  if (!hashed_password) {
    // ako se iz nekog razloga dogodi greška prilikom hashiranja lozinke
    res.status(500).send('Došlo je do greške prilikom hashiranja lozinka!');
    return;
  }

  const novi_korisnik = { id: users.length + 1, username, password: hashed_password };
  users.push(novi_korisnik);

  return res.status(201).json({ message: 'Korisnik uspješno registriran', user: novi_korisnik });
});
```

Pošaljite zahtjev preko HTTP klijenta, trebali biste dobiti odgovor u ovom obliku:

```
{
  "message": "Korisnik uspješno registriran",
  "user": {
    "id": 1,
    "username": "peroPeric123",
    "password": "$2b$10$KAPhPJRYnYZNVh.YmC3NwuaUjRPuwO.MQizgCP5kNdO/FrAa7ZXcu"
  }
}
```

Ovime smo završili prvi korak u autentifikaciji korisnika.

1.5 Provjera podudaranja *hash* vrijednosti (autentifikacija)

Recimo da se 5 korisnika registriralo u našoj aplikaciji, uključujući korisnika `peroPeric123` iz primjera iznad.

```
let users = [
    { id: 1, username: 'peroPeric123', password:
      '$2b$10$kJPhPjRyNzNVh.YmC3NwuaUjRPuW0.MQizgCP5kNdO/FrAa7ZXcu' },
    { id: 2, username: 'maraMara', password:
      '$2b$10$fnVGAkcfgSLVqGUbMGOKOU4lu3Ub6cmKyJ0aVULyK1oYOWe5MpWie' },
    { id: 3, username: 'ivanIvanko555', password:
      '$2b$10$ZKe8aSUUEBNzQlhPigzFKOBne/4v6AzEckXZ.I7.j.TXfFQRYIt8G' },
    { id: 4, username: 'anaAnic', password:
      '$2b$10$H2HR4nlpBhRFW/5YKtIuC.b5rRsPz2EE7dYz561W44/8rxJ2RrfvW' },
    { id: 5, username: 'justStanko', password:
      '$2b$10$wXcmTomNSfS9Ivafuy6/iuant3GQgxSXSwf1ZNx9d6iwuSi/d1HMK' }
];
```

Rekli smo da je matematički nemoguće "dehashirati" hash vrijednost i dobiti originalnu lozinku.

Prema tome, morat ćemo svaki put ponoviti proces hashiranja korisničke lozinke i usporediti dobiveni *hash* s onim koji je pohranjen u bazi podataka.

Međutim, potrebno je osim hashiranja ponovnog hashiranja lozinke, **provesti točan broj rundi soljenja ključa** kako bi se dobila identična *hash* vrijednost. Duljina izvođenja *hash* funkcije ovisi o broju rundi soljenja ključa, o duljini lozinke, ali i o samom algoritmu koji se koristi.

Za provjeru podudaranja hash vrijednosti sa tekstuualnom vrijednosti, koristimo metodu `compare`:

```
bcrypt.compare(plainText, hashedValue, callback);
```

Ova metoda uspoređuje `plainText` (običan tekst) s `hashedValue` (hash vrijednost) i vraća `true` ako se podudaraju, inače vraća `false`.

Rezultat funkcije je boolean vrijednost, ovisno o podudaranju hash vrijednosti s unesenom lozinkom.

Još jedanput, razlika s enkripcijom je što se **ne može "dehashirati"** hash vrijednost. Odnosno, ne možemo utvrditi podudaranje ako nemamo originalnu lozinku.

Primjer:

Obzirom da se lozinke podudaraju, očekujemo ispis Lozinke se podudaraju!.

Na isti način možemo logiku pohraniti u funkciju `checkPassword` koja prima 2 argumenta: `plainPassword` i `hashedPassword`.

```
async function checkPassword(plainPassword, hashedPassword) {
  try {
    let result = await bcrypt.compare(plainPassword, hashedPassword);
    return result;
  } catch (err) {
    console.error(`Došlo je do greške prilikom usporedbe _hash_ vrijednosti: ${err}`);
    return false;
  }
}
```

Funkciju `checkPassword` ćemo pozvati u ruti za autentifikaciju koju smo definirali na početku:

- prvo pronalazimo korisnika u listi korisnika

```
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);

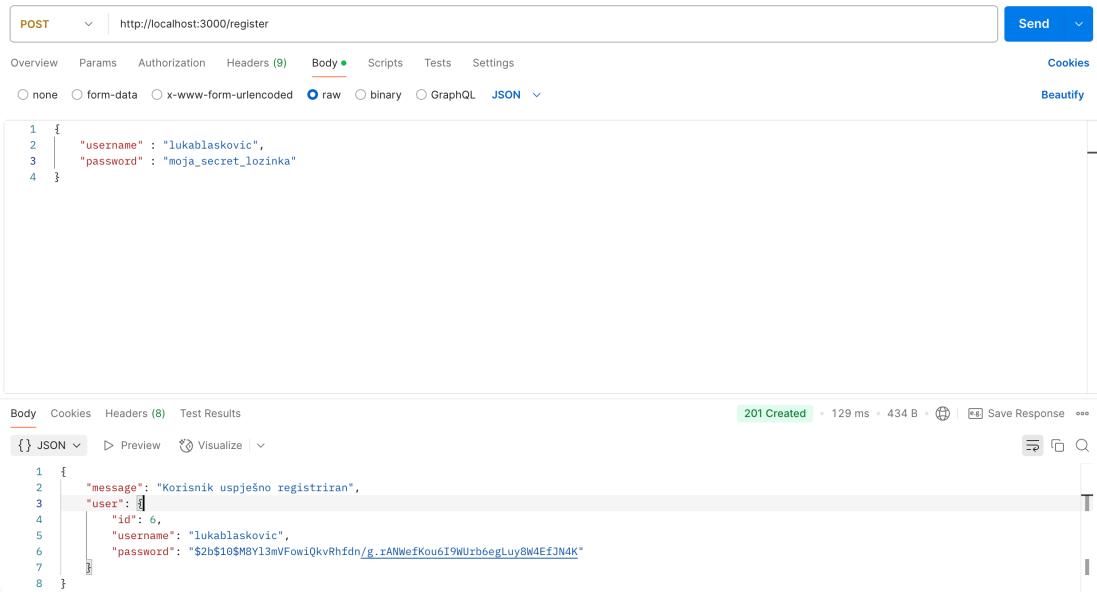
  if (!user) {
    return res.status(404).send('Ne postoji korisnik!');
  }

  const lozinkaIspravna = await checkPassword(password, user.password);

  if (lozinkaIspravna) {
    return res.send('Uspješno ste autentificirani!');
  } else {
    return res.status(401).send('Neuspješna autentifikacija!'); // 401 - Unauthorized
  }
});
```

To je to! 🚀 Testirat ćemo rute:

1. Registrirajte korisnika preko `/register` rute
2. Pokušajte autentificirati korisnika koji ne postoji preko `/login` rute
3. Autentificirajte korisnika preko `/login` rute pogrešnom lozinkom
4. Autentificirajte korisnika preko `/login` rute ispravnom lozinkom



POST http://localhost:3000/register

Body (JSON)

```

1 {
2   "username": "lukablasovic",
3   "password": "moja_secret_lozinka"
4 }

```

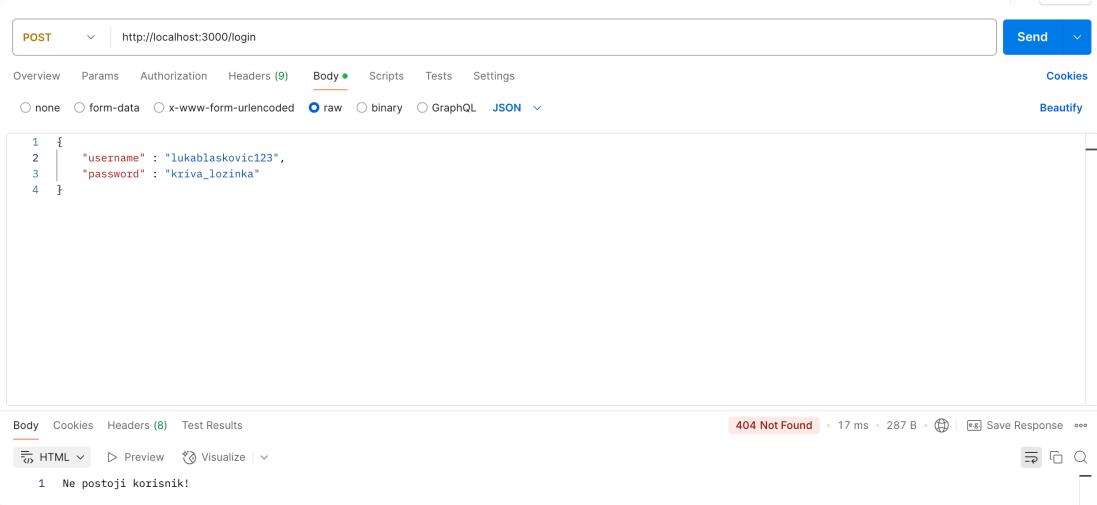
201 Created

```

1 {
2   "message": "Korisnik uspješno registriran",
3   "user": [
4     {
5       "id": 6,
6       "username": "lukablasovic",
7       "password": "$2b$10$M8Y13mVQkvRhfdn/g.rANWefKou6I9URb6egLuy8W4EfJN4K"
8     }

```

Slika 6. Registracija korisnika slanjem zahtjeva na `POST /register` u Postmanu s korisničkim imenom i lozinkom



POST http://localhost:3000/login

Body (JSON)

```

1 {
2   "username": "lukablasovic123",
3   "password": "kriva_lozinka"
4 }

```

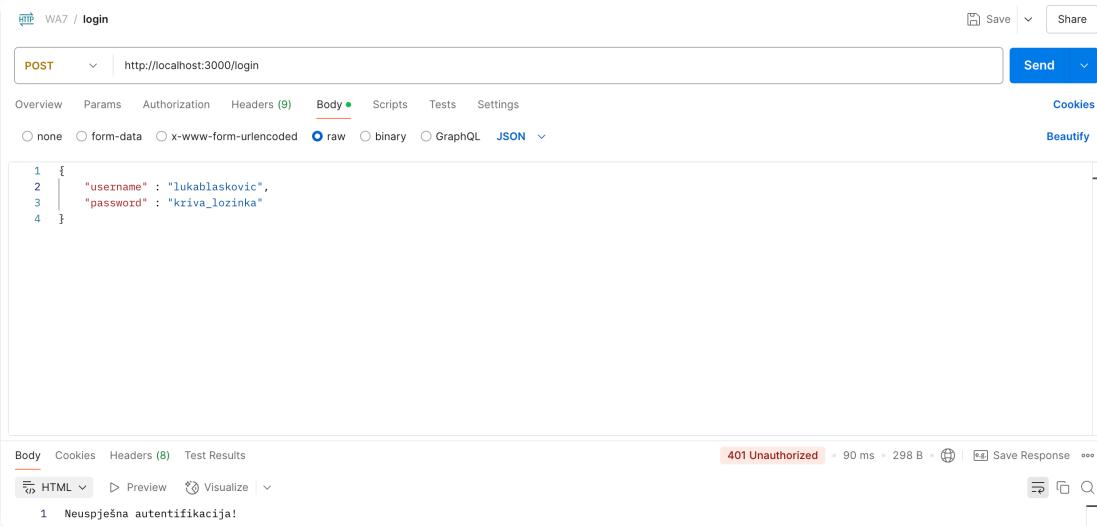
404 Not Found

```

1 Ne postoji korisnik!

```

Slika 7. Pokušaj autentifikacije nepostojećeg korisnika slanjem zahtjeva na `POST /login` u Postmanu



POST http://localhost:3000/login

Body (JSON)

```

1 {
2   "username": "lukablasovic",
3   "password": "kriva_lozinka"
4 }

```

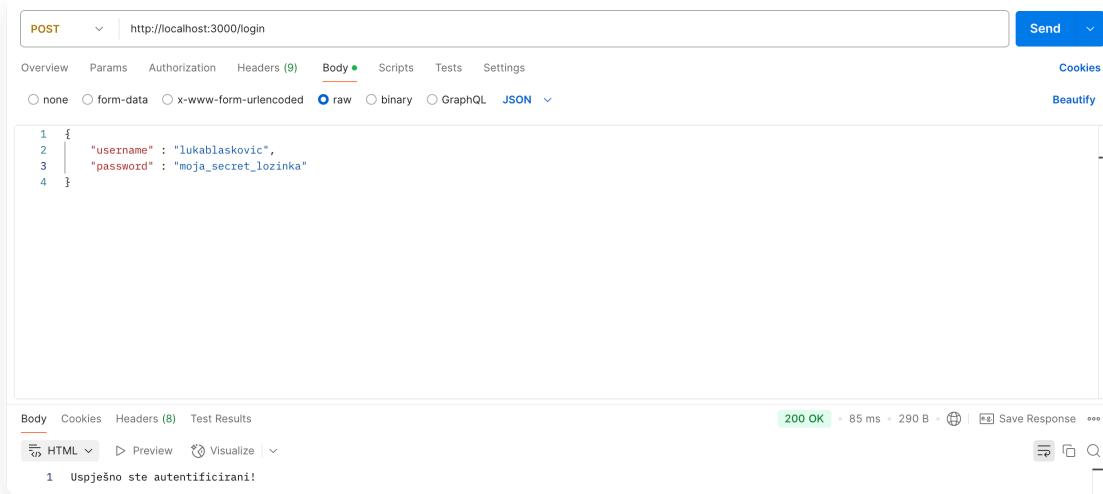
401 Unauthorized

```

1 Neuspješna autentifikacija!

```

Slika 8. Pokušaj autentifikacije postojećeg korisnika s pogrešnom lozinkom slanjem zahtjeva na `POST /login` u Postmanu



Slika 9. Uspješna autentifikacija korisnika slanjem zahtjeva na `POST /login` u Postmanu

Uspješno smo implementirali autentifikaciju korisnika putem hashiranja lozinki! 🎉

Važno je napomenuti da je ovo samo osnovna implementacija autentifikacije korisnika. U praksi, autentifikacija korisnika može biti puno složenija i uključivati dodatne sigurnosne mehanizme koje smo naveli ranije.

Dodatno, zapamtite da iz sigurnosnih razloga nije dobra praksa **slati klijentu detaljnu poruku** o grešci u slučaju neuspješne autentifikacije. Umjesto toga, preporučuje se slanje **generičke poruke o grešci** kako bi se spriječilo otkrivanje informacije što je pogrešno (korisničko ime, email ili lozinka) i zašto je došlo do greške.

Iz tog razloga, sljedeći kod:

```
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);

  if (!user) {
    return res.status(404).send('Ne postoji korisnik!');
  }

  const lozinkaIspravna = await checkPassword(password, user.password);

  if (lozinkaIspravna) {
    return res.send('Uspješno ste autentificirani!');
  } else {
    return res.status(401).send('Neuspješna autentifikacija - kriva lozinka!'); // 401 - Unauthorized
  }
});
```

- bolje je napisati ovako:

```
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);
```

```
if (!user) {
    return res.status(401).send('Neuspješna autentifikacija!'); // 401 - Unauthorized (ne
otkrivamo da korisnik ne postoji)
}

const lozinkaIspravna = await checkPassword(password, user.password);

if (lozinkaIspravna) {
    return res.send('Uspješno ste autentificirani!');
} else {
    return res.status(401).send('Neuspješna autentifikacija!'); // 401 - Unauthorized (ne
otkrivamo da je lozinka pogrešna)
}
});
```

Vidimo da u drugom primjeru, **vraćamo istu poruku o grešci** bez obzira u kojem dijelu procesa autentifikacije je došlo do greške!

2. Autorizacija kroz JWT token

Rekli smo da je **autorizacija** proces davanja prava korisniku da pristupi određenim resursima ili funkcionalnostima

U kontekstu web aplikacija, autorizaciju ćemo provoditi kroz **JWT** (*eng. JSON Web Token*).

[JSON Web Token \(JWT\)](#) je kompaktan, siguran i samodostatan način razmjene informacija između dviju strana u obliku JSON objekata. Koristi se prvenstveno za autorizaciju u modernim aplikacijama.



Slika 10. JWT Logo - JWT je službeno definiran kao standard 2015. godine kada je objavljen u [RCF 7519](#) dokumentu, međutim u razvoju je od ranih 2010-ih godina.

Prije uvođenja JWT-a kao autorizacijskog standarda, koristila se **session-based** autentifikacija koja se bazira na tzv. [HTTP kolačićima](#) (*eng. HTTP Cookies*) i **praćenju stanja korisničke sesije** na strani poslužitelja.

Web aplikacije temeljene na sesijama funkcioniраju otprilike ovako:

- poslužitelj je odgovoran za pohranu sesije korisnika u internoj memoriji ili bazi podataka
- klijent bi dobivao kolačić (*eng. cookie*) s jedinstvenim identifikatorom sesije (*eng. session ID*) nakon uspješne autentifikacije
- svaki put kada korisnik zatraži pristup resursu, kolačić se šalje natrag na poslužitelj
- poslužitelj provjerava je li sesija valjana i korisnik ima pristup resursu

Glavni nedostatak *session-based* autentifikacije jest sama činjenica da je ona *stateful* - poslužitelj mora pohranjivati i upravljati stanjem sesije korisnika. U raspodijeljenim sustavima to zahtijeva dodatne mehanizme, poput dijeljene pohrane sesija ili [sticky sessions](#), što povećava kompleksnost sustava i otežava horizontalno skaliranje aplikacije.

Kao alternativa, JWT se koristi u sklopu **token-based autentifikacije**, koja je *stateless*. Sav potreban kontekst o korisniku nalazi se unutar samog tokena, pa poslužitelji ne moraju dijeliti stanje sesije, što ovakav pristup čini prirodno pogodnim za razvoj raspodijeljenih i skalabilnih sustava.

To znači da se **svi podaci potrebni za donošenje autorizacijske odluke** nalaze u samom **tokenu**.

Ponekad se može zatražiti dodatna provjera na poslužitelju, primjerice provjera je li korisnik još uvijek aktivan ili je li token opozvan, ali u većini slučajeva, poslužitelj može donijeti odluku o autorizaciji samo na temelju podataka unutar tokena.

Značajka	Session-based	JWT (JSON Web Token)
Gdje se čuvaju podaci	Na serveru (<i>session</i>)	U tokenu kod klijenta
Što klijent šalje	Session ID (<i>cookie</i>)	JWT token (<i>header/cookie</i> - češće kroz <i>header</i>)
Server pamti stanje	Da (<i>stateful</i>)	Ne (<i>stateless</i>)
Skaliranje	Teže (<i>shared session store</i>)	Lakše (bez stanja)
Opoziv autorizacijskog pristupa	Jednostavan (obriše se <i>session</i>)	Teži (token vrijedi do isteka)
Tipična upotreba	Klasične <i>old-school</i> web aplikacije	REST-API, SPA , mobile aplikacije

2.1 Što je ustvari token?

JWT token je ništa drugo nego specijalni niz znakova koji se sastoji od 3 dijela:

1. **Header:** JSON objekt koji sadrži informacije o **tipu tokena i korištenom algoritmu za potpisivanje**
2. **Payload:** JSON objekt koji sadrži **korisničke podatke** (npr. korisničko ime, email) koje želimo "pohraniti u token", ali i **dodatne informacije** (npr. rok trajanja tokena). Payload je enkodiran [Base64URL](#) enkodingom.
3. **Signature:** kriptografski potpis generiran na temelju **headera, payloada i tajnog ili privatnog ključa**. Ovaj potpis služi za provjeru integriteta i autentičnosti tokena. Privatni/tajni ključ također može (ali i ne mora) biti enkodiran Base64URL enkodingom.

Otvorite [jwt.io](#) web aplikaciju. Ovdje možete pronaći koristan za vizualizaciju i dekodiranje JWT tokena.

Starom sučelju JWT.io aplikacije možete pristupiti preko sljedeće poveznice: <https://jwt.lannysport.net/>

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Decoded

HEADER:	<pre>{ "alg": "HS256", "typ": "JWT" }</pre>
PAYOUT:	<pre>{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }</pre>
VERIFY SIGNATURE	<pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) □ secret base64 encoded</pre>

SHARE JWT

Signature Verified

Slika 11. Primjer JWT tokena s tri dijela: **header**, **payload** i **signature**

Uočite spomenuta 3 dijela tokena, svaki je označen različitom bojom, a desno ima svoj dekodirani oblik.

1. **Header:** označen **crvenom** bojom (u ovom slučaju, koristi se `HS256` algoritam za potpisivanje, i tip tokena je `JWT`)
2. **Payload:** označen **rozom** bojom (sadrži korisničke podatke (`sub`, `name`) i `iat` - vrijeme izdavanja tokena)
3. **Signature:** označen **plavom** bojom (koristi se za provjeru integriteta tokena)

Signature kao enkripcijski algoritam koristi `HMACSHA256` (*Hash-based Message Authentication Code using SHA-256*), koji ustvari kombinira **SHA-256 hash** funkciju s **HMAC** algoritmom i sigurnim ključem za enkripciju.

U donjem lijevom kutu možemo vidjeti oznaku **Signature Verified** što znači da je token valjan i da je **integritet podataka sačuvan**.

Svaki dio JWT tokena odvojen je točkom (`.`). Ovo je važno jer nam omogućava da token dekodiramo i provjerimo njegovu valjanost.

Struktura JWT tokena:

```
header.payload.signature
```

Primjerice:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Sljedeća tablica prikazuje kodirani i dekodirani dio JWT tokena za svaki od tri dijela:

Dio JWT tokena	Kodirani dio (eng. Encoded)	Dekodirani dio (eng. Decoded)
Header	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9	{"alg": "HS256", "typ": "JWT"}
Payload	eyJzdWIoIixMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ	{"sub": "1234567890", "name": "John Doe", "iat": 1516239022}
Signature	SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c	HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret)

Ako pokušamo promijeniti bilo koji dio tokena, **signature** će se promijeniti i token više neće biti valjan; dobivamo grešku `Invalid Signature`.

Dodatno, ako promijenimo bilo koji znak u **payload** dijelu tokena, vidjet ćete u dekodiranom dijelu da se payload "pošemerno".

Encoded
Decoded

④ Invalid Signature
SHARE JWT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIoIixMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Warning: Looks like your JWT payload is not a valid JSON object. JWT payloads must be top level JSON objects as per <https://tools.ietf.org/html/rfc7519#section-7.2>

```
HEADER:
{
  "alg": "HS256",
  "typ": "JWT"
}

PAYLOAD:
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}

VERIFY SIGNATURE:
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
)
 secret base64 encoded
```

Slika 12. Promijenili smo samo jedan znak u **payload** dijelu tokena, što je rezultiralo promjenom **signature** dijela tokena i greškom `Invalid Signature`. Također, uočite kako se drastično promijenio dekodirani **payload** dio tokena.

Možemo slobodno promijeniti vrijednosti unutar **payloada**, staviti ćemo recimo `username` i `email` korisnika, a izbrisati ćemo `iat` i `sub` vrijednosti.

```
{
  "username": "peroPeric123",
  "email": "peroPeric@gmail.com"
}
```

Koji dijelovi kodiranog tokena će se sada promijeniti i zašto? 😊

► Spoiler alert! Odgovor na pitanje

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI
6IkpxVCJ9.eyJ1c2VybmcFtZSI6I
nBlcm9QZXJpYzEyMyIsImVtYWls
IjoicHBlcmt9QZXJpY0BnbWFpbC5
jb20ifQ.oyypVUgQaFNftKCPjos
TCGQK9ytP_qz_fppACfcAS8E
```

Decoded

HEADER:	{ "alg": "HS256", "typ": "JWT" }
PAYLOAD:	{ "username": "peroPeric123", "email": "peroPeric@gmail.com" }
VERIFY SIGNATURE	HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) <input type="checkbox"/> secret base64 encoded

Signature Verified

SHARE JWT

Slika 13. Promijenili smo **payload** token i došlo je do promjene **signature** dijela tokena budući da se generira na temelju **headera i payloada**

Do sad nismo koristili nikakav enkripcijski ključ za generiranje **signature** dijela tokena. **U praksi ga je potrebno koristiti kako bi se osigurala sigurnost tokena, tj. spriječilo neovlašteno mijenjanje podataka unutar tokena.**

Preporučuje se korištenje **256-bitnog ključa** za generiranje **signature** dijela tokena. Ključeve je moguće generirati pomoću raznih alata preko interneta, a možemo koristiti i `crypto` modul u Node.js-u.

```
import crypto from 'crypto';

console.log(crypto.randomBytes(32).toString('hex')) // generira 256-bitni ključ (32 x 8 = 256)
```

Ako pozovete više puta ovaj kod, svaki put ćete dobiti novi nasumični 256-bitni ključ.

Primjer generiranog ključa:

```
1b6bded687b99a58817fd80b41ca72e4dfa68087da8dac7c0a945735e525057d
```

Ključ možemo kopirati u odgovarajuće polje (dekodirani plavi dio) an jwt.io stranici.

Primjetite da se sad generira potpuno različiti **signature** dio tokena.

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.e
yJc2VybmtZSI6InBlcm9QZXJpYzEyMyIsImV
tYWlsIjoiCHB1cm9QZXJpY0BnbWFpbC5jb20if
Q.eGgt9ak7c-
SE7HnsAciey92uimRDNOH1znQiw7cJP7Y
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE
{ "alg": "HS256", "typ": "JWT" }
PAYOUT: DATA
{ "username": "peroPeric123", "email": "peroPeric@gmail.com" }
VERIFY SIGNATURE
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), 1b6bded687b99a58817fd) <input type="checkbox"/> secret base64 encoded

Signature Verified

SHARE JWT

Slika 14. Generirali smo novi **signature** dio tokena pomoću 256-bitnog ključa

Dodatno, moguće je kodirati 256-bitni ključ pomoću `base64` enkodiranja te na taj način osigurati dodatan sloj sigurnosti. `base64` nije enkripcijski algoritam, već deterministička reverzibilna funkcija koja transformira niz binarnih podataka u niz "printabilnih znakova".

2.2 Kako iskoristiti JWT token za autorizaciju?

Kako bismo koristili JWT token za autorizaciju, potrebno je:

1. **Generirati novi JWT token** prilikom **uspješne autentifikacije** korisnika koristeći korisničke podatke (npr. korisničko ime, email) i tajni ključ za potpisivanje tokena
2. **Poslati JWT token** korisniku kao odgovor u HTTP zahtjevu
3. Na korisničkoj strani, korisnik će **pohraniti JWT token** u lokalnu memoriju web preglednika koristeći `localStorage` ili `sessionStorage`.
 - `localStorage`: podaci se pohranjuju **bez vremenskog ograničenja**, odnosno ostaju pohranjeni i nakon zatvaranja preglednika ili taba, ali se **brišu čišćenjem postavki** u web pregledniku
 - `sessionStorage`: podaci se pohranjuju **samo za vrijeme trajanja sesije**, odnosno brišu se nakon zatvaranja taba ili preglednika, ili brisanjem postavki u web pregledniku
4. **Svaki put kada korisnik pristupi zaštićenom resursu**, klijentska strana **mora poslati pohranjeni JWT token** u zaglavju HTTP zahtjeva!
5. **Poslužitelj provjerava valjanost JWT tokena i dopušta pristup resursu** ako je token valjan, inače vraća autorizacijsku grešku.

Kako bismo generirali i potvrdili ispravnost JWT tokena na poslužiteljskoj strani, potrebno je koristiti `jsonwebtoken` paket.

Instalirajmo `jsonwebtoken` paket pomoću npm-a:

```
→ npm install jsonwebtoken
```

Uključimo `jsonwebtoken` u našu aplikaciju:

```
import jwt from 'jsonwebtoken';
```

JWT token generirat ćemo pomoću metode `sign`:

Sintaksa:

```
jwt.sign(payload, secretOrPrivateKey, [options, callback]);
```

gdje su:

- `payload`: JSON objekt koji sadrži korisničke podatke koje želimo pohraniti u token
- `secretOrPrivateKey`: tajni ključ koji se koristi za generiranje **signature** dijela tokena
- `options`: dodatne opcije (opcionalko) za generiranje tokena (npr. rok trajanja tokena)

Koje informacije želimo pohraniti u **payload** dijelu tokena? U pravilu, to su korisničko ime, email, ID korisnika, rola korisnika, itd. Može biti **sve od navedenog ili samo dio**. Ono što svakako nije uobičajeno, je pohranjivati **osjetljive podatke** kao što su lozinke.

Što mislite, zašto nije dobra praksa pohranjivati osjetljive podatke u JWT token, kao što su lozinke? 😬

► Spoiler alert! Odgovor na pitanje

Primjer generiranja JWT tokena:

```
let payload = { username: 'markoMaric', email: 'markooo@gmail.com' };

// random 256-bitni ključ
let secret_key = '1b6bded687b99a58817fd80b41ca72e4dfa68087da8dac7c0a945735e525057d';

let jwt_token = jwt.sign(payload, secret_key);
console.log(jwt_token);
```

Ako zalijepite ovaj kod direktno u poslužitelj, dobit ćete generirani JWT token.

Primjer generiranog JWT tokena s podacima iznad:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Im1hcmtvTWFyaWMiLCJlbWFpbCI6Im1hcmtvb29A
Z21haWwuY29tIiwiaWF0IjoxNzM2ODA1ODE1fQ.WftGGMyGh5vymH0eRz14oEpf7fPlv7Q5z0L8zoEiNdI
```

Ako zalijepite ovaj token u jwt.io aplikaciju, možete vidjeti dekodirane podatke u **payload** dijelu tokena.

Encoded PASTE A TOKEN HERE

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYOUT: DATA

```
{
  "username": "markoMaric",
  "email": "markoo@gmail.com",
  "iat": 1736805815
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) □ secret base64 encoded
```

⊗ Invalid Signature

SHARE JWT

Slika 15. Dekodirani **payload** dijelovi JWT tokena, **token nije valjan** zbog pogrešnog enkripcijiskog ključa

Međutim, vidimo da dobivamo grešku: "invalid signature". Zašto? 🤔

► Spoiler alert! Odgovor na pitanje

Zaključujemo sljedeće:

- **payload** dio tokena možemo dekodirati i vidjeti sadržaj (svatko može!)
- **signature** dio tokena ne možemo dekodirati jer ne znamo enkripcijiski ključ

Kada bi na poslužitelj stigao ovaj token, poslužitelj bi dekodirao pogrešan **signature** dio tokena i token bi bio označen kao nevaljan!

- to implicira da je klijent promijenio **signature dio tokena** i time narušio **integritet tokena**.

VAŽNO! Jedini način kako klijent može generirati ispravan **signature** dio tokena bez pomoći poslužitelja je ako sazna **enkripcijiski ključ** - što znači da je sigurnost tokena kompromitirana i mora se hitno promijeniti na poslužitelju!

Iz tog razloga, enkripcijiski ključ je potrebno pohraniti i čuvati na poslužitelju, u **varijablama okruženja** (eng. *environment variables*), a tada on postaje privatni enkripcijiski ključ (eng. *private encryption key*).

Instalirat ćemo `dotenv` paket kako bismo mogli koristiti varijable okruženja u našoj aplikaciji:

```
→ npm install dotenv
```

Uključimo `dotenv` u naš poslužitelj:

```
import dotenv from 'dotenv';

dotenv.config();
```

Izradit ćemo `.env` datoteku u korijenskom direktoriju projekta i pohraniti enkripcijiski ključ u njoj. Uobičajeno ga je nazvati `JWT_SECRET`, ali naravno, može se zvati kako god.

```
JWT_SECRET=1b6bded687b99a58817fd80b41ca72e4dfa68087da8dac7c0a945735e525057d
```

Sada možemo koristiti ovaj enkripcijski ključ u našoj aplikaciji:

```
const JWT_SECRET = process.env.JWT_SECRET;

let jwt_token = jwt.sign(payload, JWT_SECRET); // koristimo enkripcijski ključ iz varijable okruženja
console.log(jwt_token);
```

Ako unesemo ispravan enkripcijski ključ na jwt.io stranici, dobit ćemo potvrdu da je token valjan.

Encoded PASTE A TOKEN HERE

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYOUT: DATA

```
{
  "username": "markoMaric",
  "email": "markooo@mail.com",
  "iat": 1736885815
}
```

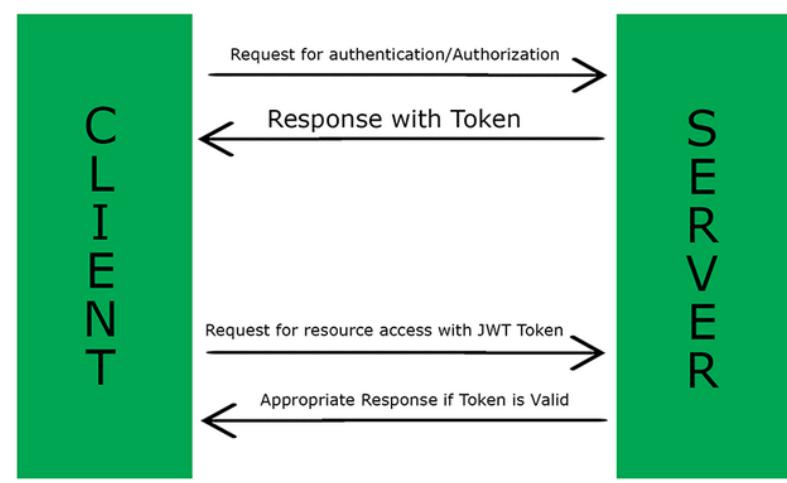
VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  dac7c0a945735e525057d
) □ secret base64 encoded
```

Signature Verified SHARE JWT

Slika 16. Dekodirani **payload** dijelovi JWT tokena s ispravnim **signature** dijelom, **token je valjan**

Ključna razlika u usporedbi s pohranom sesija i pristupom baziranim na kolačićima je što JWT token **sadrži sve informacije potrebne za autorizaciju i nema potrebe za pohranom sesije na poslužitelju**, već poslužitelj svaki put **provjerava valjanost tokena**.



Slika 17. Komunikacija između klijenta i poslužitelja koristeći JWT token za autorizaciju

2.3 Provjera valjanosti JWT tokena

Kako bismo provjerili valjanost JWT tokena na poslužiteljskoj strani, koristimo metodu `verify`:

Sintaksa:

```
jwt.verify(token, secretOrPublicKey, [options, callback]);
```

gdje su:

- `token`: JWT token koji želimo provjeriti (stijež s klijentske strane)
- `secretOrPublicKey`: tajni ključ ili javni ključ koji se koristi za provjeru **signature** dijela tokena
- `options`: dodatne opcije (opcionalno) za provjeru tokena (npr. rok trajanja tokena)
- `callback`: funkcija koja se poziva nakon provjere tokena

Callback funkcija prima dva argumenta: `err` i `decoded` gdje je `decoded` dekodirani **payload** dio tokena, a `err` je greška ako dođe do problema prilikom provjere tokena.

Primjer provjere valjanosti JWT tokena:

```
let token =
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmtZSI6Im1hcmtvTWFyaWMiLCJlbWFpbCI6Im1hcmtvb29AZ21haWwuY29tIiwiaWF0IjoxNzM2ODA2NjEwfQ.LVqWwsZBn9fxVrbeEEoKFL1VRnfnfJ2wFElpgjf2oBM';

// err je greška u slučaju da token nije valjan, decoded je dekodirani payload u slučaju da je token valjan
jwt.verify(token, JWT_SECRET, (err, decoded) => {
  if (err) {
    console.error(`Došlo je do greške prilikom verifikacije tokena: ${err}`);
    return;
  }

  console.log('Token je valjan!');
  console.log(decoded);
});
```

U konzoli dobivamo ispis:

```
Token je valjan!
{ username: 'markoMaric', email: 'markooo@gmail.com', iat: 1736806610 }
```

Ako promijenimo samo jedan znak u tokenu, dobit ćemo grešku:

```
Došlo je do greške prilikom verifikacije tokena: JsonWebTokenError: invalid token
```

2.4 Implementacija funkcija za generiranje i provjeru JWT tokena

U praksi, korisno je implementirati funkcije za generiranje i provjeru JWT tokena i smjestiti ih u zasebnu datoteku, kako bi se olakšalo njihovo korištenje u aplikaciji.

Definirajte novu datoteku `auth.js` u kojoj ćemo smjestiti sljedeće funkcije:

- `hashPassword` - funkcija za *hashiranje* lozinke
- `checkPassword` - funkcija za provjeru podudaranja lozinke i *hash* vrijednosti
- `generateJWT` - funkcija za generiranje JWT tokena **u slučaju uspješne autentifikacije**
- `verifyJWT` - funkcija za provjeru valjanosti JWT tokena **prilikom pristupa zaštićenim resursima**

Uključujemo biblioteke koje koristimo:

```
// auth.js

import bcrypt from 'bcrypt';
import dotenv from 'dotenv';
import jwt from 'jsonwebtoken';

dotenv.config();

const JWT_SECRET = process.env.JWT_SECRET;
```

Funkcija za *hashiranje* lozinke koja koristi `bcrypt` paket:

```
// auth.js

async function hashPassword(plainPassword, saltRounds) {
    try {
        let _hash_ = await bcrypt.hash(plainPassword, saltRounds); // hashiranje lozinke
        return hash;
    } catch (err) {
        console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
        return null;
    }
}
```

Funkcija za provjeru podudaranja lozinke i *hash* vrijednosti:

```
// auth.js

async function checkPassword(plainPassword, hashedPassword) {
    try {
        let result = await bcrypt.compare(plainPassword, hashedPassword); // usporedba
        lozinke i hash vrijednosti
        return result;
    } catch (err) {
        console.error(`Došlo je do greške prilikom usporedbe _hash_ vrijednosti: ${err}`);
        return false;
    }
}
```

Na isti način ćemo ukomponirati kod za generiranje JWT tokena u funkciju `generateJWT`:

```
// auth.js

async function generateJWT(payload) {
    try {
        let token = jwt.sign(payload, JWT_SECRET); // generiranje JWT tokena s payloadom i
        enkripcijskim ključem
        return token;
    } catch (err) {
        console.error(`Došlo je do greške prilikom generiranja JWT tokena: ${err}`);
        return null;
    }
}
```

I na kraju, funkcija za provjeru valjanosti JWT tokena:

```
// auth.js

async function verifyJWT(token) {
    try {
        let decoded = jwt.verify(token, JWT_SECRET); // provjera valjanosti JWT tokena
        return decoded;
    } catch (err) {
        console.error(`Došlo je do greške prilikom verifikacije JWT tokena: ${err}`);
        return null;
    }
}
```

Sada možemo koristiti ove funkcije sa svim endpointima na našem Express poslužitelju.

```
// index.js

import { hashPassword, checkPassword, generateJWT, verifyJWT } from './auth.js';
```

1. Korak (Registracija korisnika)

Prilikom registracije korisnika, koristimo funkciju `hashPassword` za *hashiranje* lozinke:

```
// index.js

app.post('/register', async (req, res) => {
  const { username, password } = req.body;

  let hashedPassword = await hashPassword(password, 10); // _hashiranje_ lozinke

  // dodajemo korisnika u listu korisnika
  users.push({ username, password: hashedPassword });

  res.status(200).send('Korisnik je uspješno registriran!');
});
```

The screenshot shows a POST request to the '/register' endpoint. The request body is a JSON object with 'username' and 'password' fields. The response shows a 200 OK status with the message 'Korisnik je uspješno registriran!'. The interface includes tabs for Body, Cookies, Headers, and Test Results.

Slika 18. (Korak 1: Registracija korisnika, šaljemo POST zahtjev na /register rutu)

2. Korak (Prijava korisnika s klijentske strane)

Nakon uspješne registracije, korisniku vraćamo potvrdu ali ne moramo vraćati hashiranu lozinku. **Želimo korisnika i hashiranu lozinku spremiti u bazu podataka.**

Radi jednostavnosti, sada ćemo ga spremiti u listu korisnika.

```
console.log(users);
```

Ispis u konzoli

```
[  
  {  
    username: 'lukablaskovic',  
    password: '$2b$10$ziHeJiULEION1DyeA5EAXOfvHnXhfGHycBJw8iyVGRa3iPA32ojhq'  
  }  
]
```

Klijentska strana obrađuje ovaj odgovor i preusmjerava korisnika na formu za prijavu.

Klijentska strana šalje POST zahtjev na /login rutu. U ovom koraku, korisnik unosi korisničko ime i lozinku.

3. Korak (Prijava korisnika na poslužiteljskoj strani)

Na poslužitelju, koristimo funkciju `checkPassword` za provjeru podudaranja lozinke i *hash* vrijednosti na početku rute `/login`.

Naravno, prvo provjeravamo postoji li korisnik u listi korisnika.

Vraćamo istu grešku ako korisnik ne postoji ili ako lozinka nije ispravna.

```
// index.js

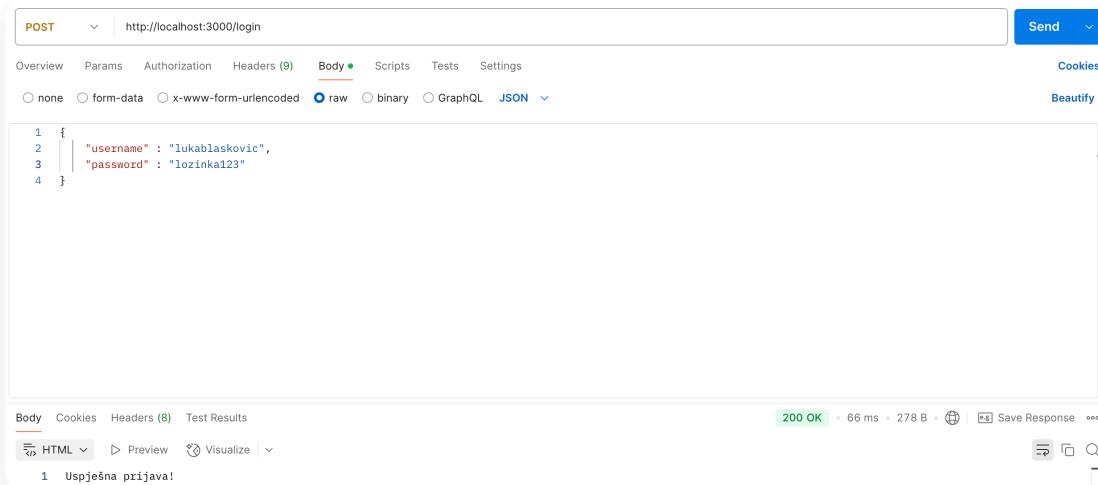
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);

  if (!user) {
    return res.status(400).send('Greška prilikom prijave!');
  }

  let result = await checkPassword(password, user.password); // usporedba lozinke i _hash_
vrijednosti

  if (!result) {
    return res.status(400).send('Greška prilikom prijave!');
  }
});
```



Slika 18. (Korak 3: Uspješna prijava korisnika, šaljemo POST zahtjev na `/login` rutu s istim podacima)

4. Korak (Generiranje JWT tokena)

Ako je prijava uspješna, šaljemo korisniku JWT token kao odgovor.

```
// index.js

app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);

  if (!user) {
```

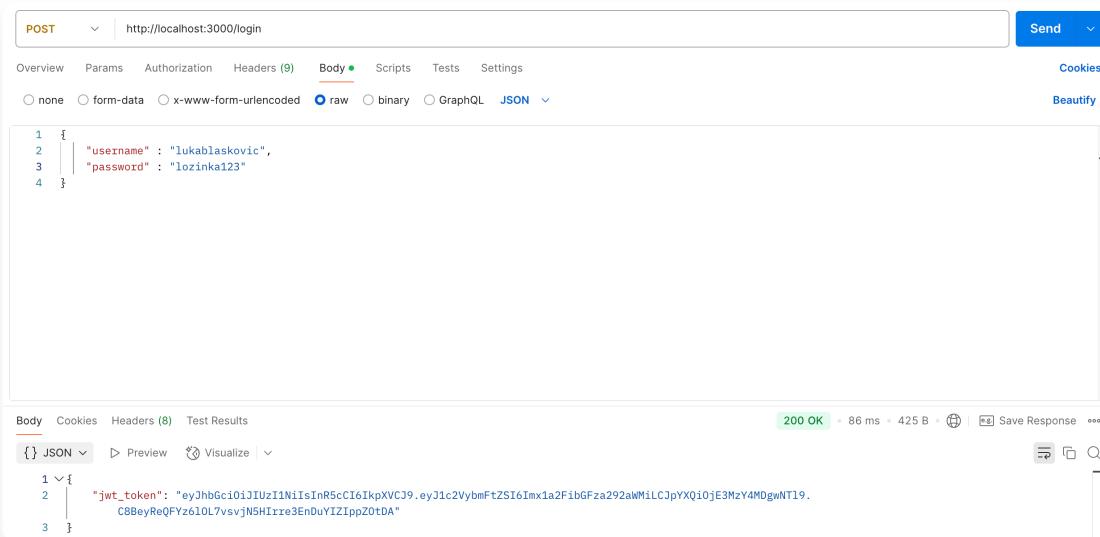
```

        return res.status(400).send('Greška prilikom prijave!');
    }

    let result = await checkPassword(password, user.password); // usporedba lozinke i _hash_
vrijednosti

    if (!result) {
        return res.status(400).send('Greška prilikom prijave!');
    }
    // ako je prijava uspješna, generiramo JWT token
    let token = await generateJWT({ id: user.id, username: user.username }); // generiranje
JWT tokena
    // šaljemo JWT token korisniku
    res.status(200).json({ jwt_token: token });
});

```



Slika 20. (Korak 4: Uspješna prijava korisnika, šaljemo JWT token kao odgovor)

5. Korak (Pohrana JWT tokena na klijentskoj strani i slanje na poslužitelj)

Korisnik sprema JWT token u lokalnu memoriju web preglednika (npr. `localStorage`).

Definirat ćemo neke resurse koji se odnose na korisnika i koji su zaštićeni, npr. možemo definirati resurs `/objave` gdje će korisnik moći pregledati samo svoje objave.

```

// index.js

let objave = [
    { id: 1, naslov: 'Prva objava', sadrzaj: 'Ovo je prva objava', autor: 'lukablaskovic' },
    { id: 2, naslov: 'Druga objava', sadrzaj: 'Ovo je druga objava', autor: 'markoMaric' },
    { id: 3, naslov: 'Treća objava', sadrzaj: 'Ovo je treća objava', autor: 'peroPeric' },
    { id: 4, naslov: 'Četvrta objava', sadrzaj: 'Ovo je četvrta objava', autor:
'lukablaskovic' }
];

```

```
// index.js

app.get('/objave', async (req, res) => {
  res.json(objave); // ali samo one koje se odnose na autoriziranog korisnika?
});
```

Rekli smo da JWT token želimo poslati u zaglavju HTTP zahtjeva: `Authorization`. Kao vrijednost ovog zaglavlja, koristimo `Bearer` prefiks i sam JWT token nakon jednog razmaka.

Bearer token predstavlja **autentifikacijski tip** koji koristi JWT token za autorizaciju.

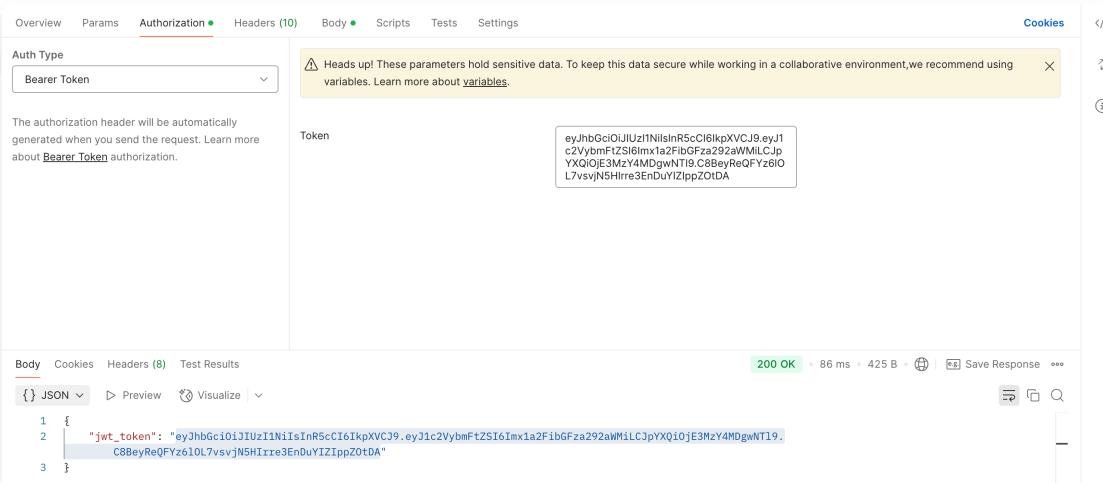
Dakle, zaglavje mora biti:

```
Authorization: Bearer <JWT token>
```

odnosno:

```
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imx1a2FibGFza292aWMiLCJpYXQiOjE3MzY4MDgwNTI9.C8BeyReQFYz6lOL7vsvjN5HIrre3EnDuYIZIppZ0tDA
```

U Postmanu je moguće odabrati tip autorizacije `Bearer Token` i zalijepiti JWT token u polje.



Slika 21. (Korak 5: Zalijepimo Bearer Token u Postmanu pod `Authorization` zaglavje)

6. Korak (Provjera valjanosti JWT tokena na poslužiteljskoj strani)

Ako se vratimo na rutu `/objave`, sada možemo provjeriti valjanost JWT tokena dohvaćanjem zaglavlja kroz `req.headers.authorization`.

- koristimo metodu `split(' ')` kako bismo odvojili `Bearer` prefiks od samog JWT tokena
- zatim se indeksiramo na sam JWT token (indeks 1)
- dekodiramo JWT token pomoću funkcije `verifyJWT` iz `auth.js` datoteke

Ako je JWT token valjan, `verifyJWT` će vratiti dekodirani **payload** dio tokena, u suprotnom će vratiti `null`.

```
// index.js
```

```

app.get('/objave', async (req, res) => {
  let token = req.headers.authorization.split(' ')[1]; // dohvaćanje JWT tokena iz zaglavlja

  let decoded = await verifyJWT(token); // provjera valjanosti JWT tokena

  if (!decoded) {
    return res.status(401).send('Nevaljan JWT token!');
  }

  // filtriramo objave prema autoru ako je JWT token valjan, odnosno ako je korisnik autoriziran
  let userObjave = objave.filter(objava => objava.autor === decoded.username); // dohvaćamo podatke iz dekodiranog payloada (decoded)

  res.json(userObjave);
});

```

Ako je JWT token valjan, **korisnik će dobiti samo one objave koje su njegove** jer smo tako definirali u funkciji `filter`.

The screenshot shows a Postman request configuration for a GET request to `http://localhost:3000/objave`. The `Authorization` tab is selected, showing `Bearer Token` as the type. A warning message states: `⚠️ Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables.`. The `Token` field contains a valid JWT token. The `Body` tab shows the response body as a JSON array:

```

[{"id": 1, "naslov": "Prva objava", "sadrzaj": "Ovo je prva objava", "autor": "lukablasovic"}, {"id": 4, "naslov": "Četvrta objava", "sadrzaj": "Ovo je četvrta objava", "autor": "lukablasovic"}]

```

The status bar at the bottom indicates a `200 OK` response with `21 ms`, `451 B`, and a `Save Response` button.

Slika 22. (Korak 6: Ako je JWT token valjan, korisnik dobiva samo svoje objave)

2.5 Autorizacijski middleware

U praksi, korisno je definirati **autorizacijski middleware** koji će se izvršiti prije svakog pristupa zaštićenim resursima kako ne bi morali svaki put provjeravati valjanost JWT tokena u samoj ruti.

Autorizacijski middleware će omogućiti da se **provjera valjanosti JWT tokena izvrši prije nego što se izvrši sama ruta**, što je u ovom slučaju poželjno ponašanje.

Definirajmo novi middleware `authMiddleware` u `auth.js` datoteci:

```
const authMiddleware = async (req, res, next) => {
    // implementacija
    next(); // nastavljamo dalje
};
```

Što ćemo staviti u ovaj middleware? **Sve što je potrebno za provjeru JWT tokena**

- pristupamo JWT tokenu iz zaglavlja (`req.headers.authorization`)
- dekodiramo JWT token pomoću funkcije `verifyJWT` iz `auth.js` datoteke
- ako je token valjan, spremamo dekodirani **payload** dio tokena u `req.authorised_user` objekt
- ako token nije valjan, vraćamo grešku

```
// auth.js

const authMiddleware = async (req, res, next) => {
    let token = req.headers.authorization.split(' ')[1]; // dohvaćanje JWT tokena iz
    zaglavlja

    let decoded = await verifyJWT(token); // provjera valjanosti JWT tokena

    if (!decoded) {
        return res.status(401).send('Nevaljan JWT token!');
    }

    req.authorised_user = decoded; // spremamo dekodirani payload u req objekt
    next(); // nastavljamo dalje
};
```

Sada možemo upotrijebiti ovaj middleware u ruti `/objave` i skratiti kod:

```
// index.js

app.get('/objave', [authMiddleware], async (req, res) => {
    let userObjave = objave.filter(objave => objave.autor === req.authorised_user.username);
    // dohvaćamo podatke iz dekodiranog payloada (req.authorised_user)

    res.json(userObjave);
});
```

Ovaj middleware možemo upotrijebiti na svim rutama koje su "zaštićene", odnosno koje zahtijevaju autorizaciju korisnika.

2.6 Rok trajanja JWT tokena

U praksi, korisno je definirati **rok trajanja JWT tokena** kako bi se sprječilo zloupotrebu tokena. Na primjer, ako maliciozni korisnik ukrade JWT token na klijentskoj strani, može ga koristiti za pristup zaštićenim resursima sve dok token ne istekne. Ako je token beskonačnog trajanja, može se koristiti zauvijek.

U tu svrhu, nije loše definirati **rok trajanja tokena** u **payload** dijelu tokena.

Prilikom generiranja JWT tokena, možemo definirati **rok trajanja** tokena u sekundama pomoću opcije `expiresIn` u `options` objektu:

```
let token = jwt.sign(payload, JWT_SECRET, { expiresIn: '1h' }); // token traje 1 sat
```

Ovaj token će trajati **1 sat** od trenutka generiranja. Nakon toga, funkcija `verify` će vratiti grešku `"TokenExpiredError"`. U tom slučaju potrebno je na klijentskoj strani preusmjeriti korisnika na formu za prijavu, a poslužitelj neće dozvoliti pristup zaštićenim resursima.

Samostalni zadatak za Vježbu 7

Nadogradite *pizza-poslužitelj* s prethodnih vježbi autentifikacijom korisnika i autorizacijom zahtjeva koristeći JWT token.

1. Implementirajte registraciju korisnika na poslužiteljskoj strani, a na klijentskoj strani omogućite korisniku unos korisničkog imena i lozinke.
2. Na poslužiteljskoj strani pohranite korisnika u Mongo bazu, a lozinku obvezno *hashirajte* prije pohrane koristeći `bcrypt` paket.
3. Na poslužiteljskoj strani implementirajte rutu za prijavu korisnika, gdje korisnik unosi korisničko ime i lozinku a dobiva potpisani JWT token koji traje 24 sata.
4. Implementirajte autorizacijski middleware koji će izvršiti prije rute za **dohvaćanje svih narudžbi korisnika** te za **slanje nove narudžbe**. Nakon provjere ispravnosti JWT tokena, moraju biti vraćene one narudžbe koje se odnose na autoriziranog korisnika odnosno bez autorizacija - korisnik ne može poslati novu narudžbu. Ako token nije valjan, vratite odgovarajuću grešku.
5. Na klijentskoj strani implementirajte pohranu JWT tokena u `localStorage` i slanje tokena u zaglavljtu HTTP zahtjeva na svaku rutu koja zahtijeva autorizaciju, npr. dohvaćanje svih zadataka. Koristite paket `axios` za slanje HTTP zahtjeva i dodajte mu zaglavlje `Authorization` s `Bearer` tokenom.

U zadaći je potrebno učitati frontend i backend kod iz prethodnih vježbi i nadograditi ih s funkcionalnostima autentifikacije i autorizacije.