Razvoj decentralizirane aplikacije koristeći pametne ugovore

Šajfar, Sanja

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:270680

Rights / Prava: Attribution-NonCommercial 3.0 Unported/Imenovanje-Nekomercijalno 3.0

Download date / Datum preuzimanja: **2024-12-27**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Sanja Šajfar

Razvoj decentralizirane aplikacije koristeći pametne ugovore

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Sanja Šajfar

Matični broj: 0016136192

Studij: Informatika u obrazovanju

Razvoj decentralizirane aplikacije koristeći pametne ugovore

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Neven Vrček

Varaždin, srpanj 2024.

Sanja Šajfar

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Diplomski rad istražuje proces razvoja i implementacije pametnih ugovora na Ethereum platformi. Rad počinje s teorijskim uvodom u blockchain tehnologiju s naglaskom na funkcionalnost i primjenu pametnih ugovora. Kroz korištenje alata kao što su programski jezik Solidity za pisanje pametnih ugovora i React za razvoj korisničkog sučelja, istraženi su ključni elementi potrebni za izradu decentraliziranih aplikacija. U praktičnom dijelu rada razvijen je decentralizirani webshop koji omogućava kupnju proizvoda korištenjem kriptovalute Ethereum (ETH). Metamask je korišten za upravljanje Ethereum novčanikom, dok je Hardhat služio za razvoj i testiranje pametnih ugovora. Na kraju rada, dan je zaključak i osvrt na rad

Ključne riječi: blockchain, pametni ugovori, Ethereum, Solidity, React, Metamask, Hardhat

Sadržaj

1. Uvod

U današnjem digitalnom dobu internet i tehnologija igraju ključnu ulogu u svakodnevnom životu, poslovanju i komunikaciji. Tehnološki napredak omogućio je razvoj novih i inovativnih rješenja koja mijenjaju način na koji obavljamo transakcije, dijelimo informacije i surađujemo. Jedna od najznačajnijih tehnologija koja se pojavila u posljednjem desetljeću je blockchain ili lanac blokova. Ova tehnologija omogućuje sigurne, transparentne i decentralizirane transakcije bez potrebe za posrednicima.

Decentralizirane aplikacije (dApps) predstavljaju jedan od najznačajnijih pomaka u tehnologiji lanca blokova jer nude rješenja koja omogućuju interakciju između korisnika bez potrebe za posrednicima. Razvoj ovih aplikacija, temeljenih na pametnim ugovorima, omogućuje transparentne, sigurne i nepromjenjive transakcije koje su potpuno automatizirane. Ovaj diplomski rad istražuje proces razvoja decentralizirane aplikacije koristeći pametne ugovore na Ethereum platformi s naglaskom na stvaranje webshopa koji omogućuje kupnju proizvoda korištenjem kriptovalute Ethereum (ETH).

Rad započinje teorijskim pregledom tehnologije lanca blokova, objašnjavajući njezine ključne komponente, vrste lanaca blokova i osnovne principe na kojima se temelji. Poseban fokus stavljen je na Ethereum kao jednu od najznačajnijih platformi za razvoj dApps-a, objašnjavajući funkcionalnosti Ethereum Virtual Machine (EVM) i napredak prema Ethereum 2.0. U teorijskom dijelu također se detaljno razmatraju pametni ugovori, njihova povijest, ključne karakteristike i pravni izazovi.

Praktični dio rada uključuje razvoj webshopa koristeći programski jezik Solidity za pisanje pametnih ugovora i React za razvoj korisničkog sučelja. Implementirani primjer služi kao dokaz koncepta, demonstrirajući osnovne funkcionalnosti dApps-a, uključujući kreiranje i izvršavanje transakcija putem pametnih ugovora.

2. Blockchain tehnologija

Da bismo razumjeli arhitekturu lanca blokova, prvo moramo shvatiti što je tehnologija lanca blokova. Lanac blokova je u osnovi lanac blokova koji sadrže specifične informacije (baza podataka), ali na siguran i autentičan način koji je grupiran zajedno u mreži (peer-to-peer). Drugim riječima, lanac blokova je kombinacija računala povezanih jednih s drugima umjesto s centralnim serverom, što znači da je cijela mreža decentralizirana [1].

Pojednostavljeno, koncept lanca blokova može se usporediti s radom na zajedničkim tablicama u Excelu. U prošlosti smo morali slati tablice naprijed-nazad putem emaila, što je dovodilo do zbrke i različitih verzija istog dokumenta. Danas uz pomoć online alata poput Google Sheets svi sudionici mogu istovremeno raditi na istoj tablici, a promjene su odmah vidljive svima. Tehnologija lanca blokova omogućuje distribuciju digitalnih informacija umjesto njihovog kopiranja, pružajući transparentnost, povjerenje i sigurnost podataka. Lanac blokova se široko koristi u financijskoj industriji te danas se ova tehnologija koristi za stvaranje softverskih rješenja za kriptovalute, vođenje evidencije i pametne ugovore.

2.1. Vrste lanca blokova

Tehnologija lanaca blokova se razvila kako bi zadovoljila različite potrebe i primjene. Prema strukturi i načinu pristupa, razlikujemo četiri glavne vrste lanaca blokova: javni, privatni, konzorcijski i hibridni lanac blokova [1] [2].

Javni lanac blokova je otvoren za sve korisnike s pristupom internetu. Ova vrsta omogućuje svakom korisniku da postane čvor, pregledava prošle zapise i validira nove transakcije. Javni lanci blokova pružaju visoku razinu transparentnosti i decentralizacije što ih čini otpornima čak i ako izvorna organizacija prestane s radom. Glavna prednost je visoka razina sigurnosti i nepromjenjivosti, iako su manje efikasni i zahtijevaju značajne računalne resurse. Primjeri takvog lanca blokova bi bili Bitcoin i Ethereum [1] [2].

Privatni lanac blokova se koristi u ograničenim okruženjima, često pod kontrolom jedne organizacije. Iako i dalje održava *peer-to-peer* komunikaciju i određeni stupanj decentralizacije, pristup je ograničen na autorizirane sudionike. Ova vrsta lanca blokova obično se koristi unutar organizacija za interne procese. Ključne prednosti uključuju poboljšane sigurnosne postavke i kontrole pristupa, ali centralizacija ga čini podložnijim potencijalnim manipulacijama u usporedbi s javnim lancima blokova [1] [2].

Konzorcijski lanac blokova kombinira značajke javnih i privatnih lanaca blokova. Njime upravlja skupina organizacija, što osigurava decentraliziraniji pristup u odnosu na

privatne lance blokova. Ova vrsta omogućuje da određeni podaci budu javno dostupni dok su drugi podaci ograničeni. Konzorcijski lanci blokova su posebno korisni u suradničkim okruženjima gdje više organizacija treba raditi zajedno s dijeljenom infrastrukturom lanca blokova [1] [2].

Hibridni lanac blokova kombinira elemente javnih i privatnih lanaca blokova kako bi ponudio fleksibilnost u kontroli pristupa. Ova vrsta omogućuje da neki podaci budu javni dok su drugi podaci privatni i dostupni samo autoriziranim čvorovima. Hibridni lanci blokova su idealni za scenarije gdje je potrebna javna transparentnost za neke informacije, dok se osjetljivi podaci moraju održavati privatnima. Ova struktura nastoji balansirati između transparentnosti, sigurnosti i efikasnosti [1] [2].

Postoji nekoliko ključnih razlika između različitih vrsta lanaca blokova koje utječu na njihovu primjenu, sigurnost, efikasnost i stupanj decentralizacije. Te razlike uključuju [1] [2]:

- **Proces konsenzusa:** Javni lanac blokova omogućuje svim čvorovima da sudjeluju u procesu konsenzusa, dok je u privatnom lancu blokova konsenzus kontroliran od strane jedne organizacije. Konzorcijski lanac blokova postiže konsenzus kroz odabranu grupu organizacija, dok u hibridnom lancu blokova sudjeluje odabrani set čvorova.
- **Pristupne dozvole**: U javnom lancu blokova pristupne dozvole su otvorene za sve, dok su u privatnom lancu blokova ograničene na autorizirane korisnike. Konzorcijski lanac blokova može imati javni ili ograničeni pristup, ovisno o postavkama, dok hibridni lanac blokova omogućuje javni pristup za neke podatke i ograničeni za druge.
- **Nepromjenjivost**: Što se tiče nepromjenjivosti, javni lanac blokova je visoko nepromjenjiv zbog široke distribucije, dok je privatni lanac blokova lako promjenjiv zbog centralne kontrole. Konzorcijski lanac blokova je nepromjenjiviji od privatnog, ali manje od javnog, dok hibridni lanac blokova nudi djelomičnu nepromjenjivost ovisno o vrsti podataka i pristupu.
- **Efikasnost**: Efikasnost javnog lanca blokova je općenito niža zbog velikog broja čvorova i računalnih zahtjeva, dok su privatni lanci blokova efikasniji s bržim vremenima transakcija. Konzorcijski lanac blokova nudi visoku efikasnost, balansirajući decentralizaciju i brzinu, dok hibridni lanac blokova nudi efikasnost sličnu privatnim lancima blokova uz zadržavanje prednosti javnog pristupa.
- **Centralizacija** : Javni lanac blokova je potpuno decentraliziran, privatni lanac blokova je potpuno centraliziran, konzorcijski lanac blokova je djelomično centraliziran, kontroliran od strane više entiteta, dok hibridni lanac blokova balansira kontrolu i decentralizaciju.

2.2. Arhitektura lanca blokova

Tradicionalna arhitektura svjetske mreže koristi klijent-server mrežu, gdje server čuva sve potrebne informacije na jednom mjestu. Ovo olakšava ažuriranje jer serverom upravlja ograničeni broj administratora s dozvolama. Nasuprot tome, u decentraliziranoj mreži lanca blokova svaki sudionik unutar mreže održava, odobrava i ažurira nove zapise. Sustav nije kontroliran samo od strane pojedinaca, već od strane svih unutar mreže lanca blokova. Svaki član osigurava da su svi zapisi i postupci ispravni, što rezultira valjanošću i sigurnošću podataka. Na taj način, strane koje ne moraju nužno imati povjerenja jedna u drugu mogu postići zajednički konsenzus [2].

Osnovne komponente arhitekture lanca blokova

- **Čvor**: korisnik ili računalo unutar arhitekture lanca blokova (svaki ima neovisnu kopiju cijelog registra lanca blokova)
- **Transakcija**: najmanji građevni blok sustava lanca blokova (zapisi, informacije, itd.) koji služi svrsi lanca blokova
- **Blok**: podatkovna struktura koja se koristi za čuvanje skupa transakcija koje se distribuiraju svim čvorovima u mreži
- **Lanac**: niz blokova u specifičnom redoslijedu
- **Rudari**: specifični čvorovi koji obavljaju proces verifikacije blokova prije nego što se bilo što doda strukturi lanca blokova
- **Konsenzus**: skup pravila i dogovora za provođenje operacija lanca blokova

U nastavku će neke od komponenata arhitekture lanca blokova biti detaljnije objašnjene.

Blokovi su osnovne jedinice u lancu blokova, a svaki blok sadrži skup transakcija. Blokovi su međusobno povezani i formiraju lanac, gdje svaki blok sadrži podatke o transakcijama, hash prethodnog bloka koji osigurava povezanost između blokova i sprječava izmjene u prethodnim blokovima te hash trenutnog bloka, jedinstveni identifikator bloka generiran na temelju njegovog sadržaja. Sve te informacije koje svaki blok sadrži su potrebne kako bi se osiguralo da su svi podaci unutar lanca blokova sigurni i otporni na manipulacije.

Kriptografija je tu najbitnija komponenta koja dodatno osigurava sigurnost i integritet podataka u lancu blokova. Hash funkcije koriste se za stvaranje jedinstvenih identifikatora za svaki blok i transakciju. One osiguravaju da i najmanja promjena u podacima rezultira potpuno različitim hashom, što omogućuje otkrivanje bilo kakvih izmjena. Digitalni potpisi koriste se za

autentifikaciju i verifikaciju identiteta korisnika, osiguravajući da transakcije potpisuju legitimni vlasnici privatnih ključeva.

Nadalje, lanac blokova funkcionira unutar decentralizirane peer-to-peer (P2P) mreže gdje svaki čvor (računalo) u mreži ima kopiju cijelog lanca blokova. P2P mreža osigurava distribuirano pohranjivanje podataka, otpornost na kvarove i distribuirano procesiranje. Svi čvorovi imaju identične kopije lanca blokova, čime se eliminira potreba za centralnim poslužiteljem. Čak i ako neki čvorovi ne uspiju, mreža kao cjelina ostaje operativna. Validacija transakcija i stvaranje novih blokova provode se distribuirano među čvorovima čime se dodatno osigurava integritet i sigurnost mreže.

Za sigurnost i valjanost su vrlo bitni i konsenzusni algoritmi. Oni omogućuju čvorovima u mreži da se slože o valjanosti transakcija i redoslijedu blokova. Dva najčešće korištena konsenzusna algoritma su *Proof of Work* (PoW) i *Proof of Stake* (PoS). U PoW algoritmu, čvorovi (rudari) rješavaju složene matematičke probleme kako bi stvorili nove blokove. PoW je energetski intenzivan, ali osigurava visoku razinu sigurnosti. PoS algoritam bira čvorove za stvaranje novih blokova na temelju količine kriptovaluta koju drže i voljni su staviti kao zalog, što je energetski učinkovitije od PoW-a. Ovi algoritmi osiguravaju da se sve transakcije unutar mreže valjano verificiraju i dodaju u lanac blokova. Pametni ugovori su još jedan ključni element arhitekture lanca blokova. O njima će biti više riječi u sljedećem poglavlju.

Arhitektura lanca blokova pruža snažnu osnovu za sigurnu, transparentnu i decentraliziranu pohranu i prijenos podataka. Ova tehnologija ima široku primjenu u financijama, logistici, zdravstvenoj skrbi, energetici i mnogim drugim industrijama. Lanac blokova pruža sigurne i učinkovite metode za upravljanje podacima i transakcijama te je od robusnijih i fleksibilnijih rješenja za brojne izazove modernog poslovanja i tehnologije. U nastavku će biti više riječi o Ethereum lancu blokova.

3. Ethereum lanac blokova

Ethereum je decentralizirana platforma koja omogućuje izradu i implementaciju pametnih ugovora i decentraliziranih aplikacija (dApps). Ethereum je izumljen s ciljem da se proširi mogućnosti koje je originalno pružio Bitcoin, uvodeći lanac blokova koji omogućuje stvaranje složenih aplikacija. Ethereum je osmišljen 2013. godine, a službeno je pokrenut 2015. godine od strane Ethereum Foundation. Glavna razlika između Ethereuma i Bitcoina je u tome što je Ethereum mnogo više od kriptovalute, to je platforma koja omogućuje izradu decentraliziranih aplikacija koristeći lanac blokova. Bitcoin je prvenstveno osmišljen kao digitalna valuta za razmjenu dobara i usluga, dok Ethereum omogućuje stvaranje aplikacija koje koriste lanac blokova za pohranu podataka i izvršavanje funkcija [3].

Ethereum koristi vlastitu kriptovalutu pod nazivom Ether (ETH), koja služi kao gorivo za mrežu. Ether se koristi za plaćanje transakcijskih naknada i računalnih usluga na mreži Ethereum. Svaka transakcija ili izvršenje pametnog ugovora zahtijeva određenu količinu Etera, koja se mjeri u jedinicama nazvanim "gas".

3.1. Ethereum Virtual Machine (EVM)

Jedna od ključnih komponenti Ethereuma je Ethereum Virtual Machine (EVM). EVM je decentralizirano računalno okruženje koje izvršava pametne ugovore. Svaki čvor na mreži Ethereum pokreće kopiju EVM-a, omogućujući izvršavanje istog koda na isti način na svim čvorovima. Ovo osigurava da su rezultati izvršenja identični na cijeloj mreži [3].

EVM omogućuje programerima da kreiraju aplikacije koristeći različite programske jezike, uključujući Solidity, Vyper i druge. Solidity je najpopularniji programski jezik za razvoj pametnih ugovora na Ethereumu te o njemu će se moći pročitati više u kasnijem poglavlju. EVM osigurava da se svi pametni ugovori izvršavaju na isti način, bez obzira na čvor na kojem se izvršavaju. Ovo osigurava dosljednost i pouzdanost mreže [3] [4].

Nadalje, jedna od ključnih prednosti EVM-a je njegova *Turing-complete* priroda, što znači da može izvršavati bilo koju računalno uračunljivu funkciju. Ovo omogućuje programerima da kreiraju složene aplikacije koje mogu obavljati razne zadatke, od financijskih transakcija do upravljanja podacima. EVM također omogućuje izolaciju pametnih ugovora. To znači da jedan ugovor ne može izravno utjecati na drugi te se na taj način povećava sigurnost i stabilnost mreže. Nadalje, EVM koristi koncept stanja (eng. *state*) za praćenje svih transakcija i pametnih ugovora na mreži. Svaki račun na mreži Ethereum ima stanje koje uključuje saldo Etera, kod pametnog ugovora i pohranjene podatke. Kada se izvrši transakcija ili pametni

ugovor, stanje se ažurira na svim čvorovima u mreži kako bi se osigurala konzistentnost [3] [4].

3.2. Ethereum 2.0

Ethereum 2.0, poznat i kao Eth2, izvorno se koristio za razlikovanje Ethereum mreže prije i nakon prelaska na proof-of-stake ili ponekad za označavanje različitih Ethereum klijenata (klijenti za izvršavanje su ponekad nazivani ETH1 klijentima, dok su klijenti za konsenzus nazivani ETH2 klijentima). Ethereum 2.0 obuhvaća niz ažuriranja dizajniranih za rješavanje problema ograničene skalabilnosti, brzine i ograničenja *Proof of Work* (PoW) konsenzusnog mehanizma na Ethereum mreži. Ove ideje dolaze od Vitalika Buterina, osnivača Ethereuma, i istraživača Vlada Zamfira. Ethereum 2.0 ima za cilj riješiti probleme skalabilnosti i visokih transakcijskih naknada koji su postali očiti kako je popularnost mreže rasla. Uvođenjem PoS i shardinga, Eth2 je omogućio mreži da podrži više korisnika i transakcija bez kompromitiranja performansi ili sigurnosti [5] [6].

Implementacija Ethereum 2.0 nije se dogodila odjednom, već je izvedena u fazama kako bi prijelaz bio što glađi i sigurniji. Proces je obuhvatio tri glavne faze koje su se provodile od 2020. do 2023. godine. Prva faza, nazvana Faza 0, započela je 1. prosinca 2020. godine lansiranjem Beacon Chain-a, koji je funkcionirao paralelno s glavnom Ethereum mrežom. Ova nova komponenta uvela je *Proof of Stake* (PoS) u Ethereum ekosustav, omogućujući korisnicima da stake-aju ETH. Ipak, povlačenje uloga nije bilo moguće sve do dovršetka Faze 2 u travnju 2023 [5].

Nakon lansiranja Beacon Chain-a, uslijedila je Faza 1 koja je uključivala uvođenje shard lanaca. Umjesto da cijeli ekosustav radi na jednoj glavnoj lanac blokova mreži, opterećenje je raspodijeljeno na 64 shard lanca kojima je koordinirao Beacon Chain. *Shard* je zaseban lanac blokova unutar ekosustava koji je povezan s glavnim lancem blokova i drugim shardovima [6].

Shard lanci omogućuju bržu validaciju transakcija jer se validatori fokusiraju samo na jedan shard, umjesto na cijeli lanac blokova. Ovo je povećalo skalabilnost mreže, omogućujući obradu većeg broja transakcija i smanjenje hardverskih zahtjeva za validatore. U budućnosti, ovo može omogućiti vođenje Ethereum mreže na mobilnim uređajima ili prijenosnim računalima.

Faza 2, također poznata kao Spajanje, dovršena je djelomično u rujnu 2022. godine. Tijekom ovog procesa, glavna Ethereum mreža spojila se s Beacon Chain-om, čime je dovršena nadogradnja Ethereum 2.0. Spajanjem je *Proof of Work* algoritam eliminiran iz

Ethereum ekosustava, a *Proof of Stake* je preuzeo punu ulogu u mreži. Nakon spajanja, Ethereum je ponovno omogućio pokretanje pametnih ugovora na obnovljenoj mreži. Iako je prijelaz na PoS dovršen, tek s nadogradnjom Shanghai u travnju 2023. korisnici su mogli povući svoje uložene fondove [5] [6].

Prije Ethereum 2.0, mreža je mogla obraditi između 15 i 45 transakcija po sekundi. Uvođenje 64 *shard* lanaca značajno je smanjilo vrijeme potrebno za validaciju transakcija, čime se poboljšala skalabilnost i brzina mreže [5].

3.3. Rizici i izazovi

Iako Ethereum pruža mnoge prednosti, postoje i rizici i izazovi povezani s njegovom upotrebom. Jedan od glavnih rizika je sigurnost pametnih ugovora. Budući da su pametni ugovori nepromjenjivi jednom kada su postavljeni na lanac blokova bilo kakve pogreške u kodu mogu imati ozbiljne posljedice. To zahtijeva da programeri pažljivo pregledaju i testiraju svoj kod prije implementacije. Nadalje, postoji stalna prijetnja od hakiranja i drugih oblika cyber napada, koji mogu ugroziti sigurnost mreže i korisnika.

Jedan od najslavnijih primjera sigurnosnog propusta bio je DAO (eng. *Decentralized Autonomous Organization*) hack 2016. godine, gdje su hakeri iskoristili ranjivost u pametnom ugovoru i ukrali oko 50 milijuna dolara u Eteru. Ovaj incident doveo je do podjele Ethereum mreže na dvije verzije: Ethereum (ETH) i Ethereum Classic (ETC). Ova podjela pokazuje koliko su važni sigurnosni protokoli i pažljivo testiranje pametnih ugovora. Nadalje, drugi primjer je incident iz 2018. godine, kada je jedna australska tvrtka izgubila 6,6 milijuna dolara zbog "backdoor" funkcije u SoarCoin ugovoru. Kako bi se zaštitili od ovih prijetnji, korisnici dApp aplikacija trebaju pažljivo pregledavati funkcije prijenosa, generiranja ili uništavanja tokena unutar pametnih ugovora. Ako ove funkcije mogu izvršavati samo specifični računi i utjecati na stanje drugih računa, postoji rizik od zlouporabe [7].

S uvođenjem novih ažuriranja poput Ethereum 2.0 Ethereum mreža se suočava s raznim sigurnosnim izazovima. Prijelaz s *Proof of Work* (PoW) na *Proof of Stake* (PoS) donio je značajna poboljšanja u pogledu energetske učinkovitosti i sigurnosti. PoS smanjuje rizik od 51% napada. PoS smanjuje potrošnju energije za oko 99,95% jer eliminira potrebu za rudarenjem. Ova promjena također drastično smanjuje potrošnju energije, čineći Ethereum ekološki prihvatljivijim [7].

Jedan od glavnih razloga zašto Ethereum ima potencijalnu svijetlu budućnost je njegova sposobnost da podrži širok spektar aplikacija. Od decentraliziranih financijskih aplikacija (DeFi) do igara, društvenih mreža i drugih dApps, Ethereum nudi platformu koja

omogućuje inovacije i kreativnost. Njegova otvorena priroda i snažna zajednica osiguravaju da će Ethereum nastaviti rasti i razvijati se. Osim tehničkih poboljšanja, Ethereum ima i jaku podršku zajednice, koja uključuje programere, investitore i poduzetnike širom svijeta.

4. Pametni ugovori

U ovom poglavlju ću prvo iznijeti kratku povijest kako su nastali pametni ugovori te zatim ću dati definiciju pametnih ugovora, objasniti njihove tehničke aspekte te ključne karakteristike. Zatim ću se osvrnuti na pravne implikacije. Nadalje ću kroz primjere iz stvarnog života prikazat praktičnu primjenu pametnih ugovora u različitim industrijama. Na kraju, razmotrit ću potencijalni budući razvoj ove tehnologije.

4.1. Povijest pametnih ugovora

Kako bismo bolje razumjeli što su pametni ugovori danas, kako se koriste i čemu služe, potrebno je vidjeti kako su se oni razvili, njihovu povijest. Prvi koraci prema pametnim ugovorima započeli su početkom 1990-ih. Godine 1991., S. Haber i W.S. Stornetta osmislili su sustav koji je bio otporan na manipulacije i omogućavao vremensko označavanje digitalnih dokumenata. Njihov sustav je dokumentima dodjeljivao certifikat koji je sadržavao datum stvaranja i informacije o prethodno izdanim certifikatima, čime se stvarala poveznica koja je mogla poslužiti za dokazivanje vremena stvaranja dokumenta [8] [9].

Nick Szabo je 1994. godine prvi puta predstavio koncept pametnih ugovora, računalnih programa koji repliciraju radnje opisane u tradicionalnim ugovorima. Kasnije, 1996. godine, Szabo je definirao ciljeve pametnih ugovora: uočljivost, provjerljivost, povjerljivost i provedivost [9].

Međutim, stvarni zamah prema onome što danas poznajemo kao lanac blokova dogodio se 2008. godine, kada je Satoshi Nakamoto predstavio Bitcoin. Bitcoin je decentralizirani sustav plaćanja gdje se povijest transakcija pohranjuje u distribuiranu knjigu sastavljenu od blokova. Svaki blok sadrži skup transakcija, vremensku oznaku i hash prethodnog bloka, čime se stvara neprekinuti lanac podataka. Ovaj sustav je pokazao kako se može eliminirati potreba za centraliziranim institucijama poput banaka, omogućujući korisnicima prijenos valuta i sudjelovanje u procesu verifikacije transakcija. Iako Bitcoin skriptni jezik omogućuje samo provjeru i validaciju transakcija, može se smatrati prvom implementacijom pametnih ugovora na lancu blokova [8] [9].

Pravi napredak u razvoju pametnih ugovora dogodio se 2015. godine kada je Vitalik Buterin stvorio Ethereum, lanac blokova platformu koja podržava decentralizirani sustav plaćanja i jezik nazvan Solidity. Ovo je omogućilo razvoj širokog spektra pametnih ugovora na blockchain-u koji se mogu lako provjeravati i ovisno o načinu pristupa lancu blokova osigurati povjerljivost [8] [9].

Može se vidjeti da povijest pametnih ugovora pokazuje značajan napredak. Od prvih koncepata vremenskog označavanja digitalnih dokumenata do današnjih rješenja baziranih na lancu blokova, pametni ugovori su evoluirali i imaju potencijal značajno promijeniti način na koji provodimo poslovne transakcije i ugovorne odnose. Pametni ugovori danas nalaze primjenu u raznim industrijama, uključujući financije i zdravstvenu zaštitu, omogućujući automatsko izvršenje ugovornih obveza bez potrebe za posrednicima. Iako su još uvijek u ranoj fazi razvoja, njihov potencijal za transformaciju tradicionalnih ugovornih odnosa je izniman.

4.2. Definicija i ključne karakteristike pametnih ugovora

Pametni ugovori su računalni programi koji automatski izvršavaju unaprijed definirane radnje kada su ispunjeni određeni uvjeti. Njihova primarna svrha je eliminirati potrebu za posrednicima i osigurati da se ugovorne obveze provode točno kako je dogovoreno. Kao što je već bilo spomenuto, koncept pametnih ugovora prvi je put predstavio Nick Szabo 1994. godine. Szabo ih je definirao kao digitalne protokole za informiranje, potvrđivanje i izvršavanje uvjeta ugovora. Njegova vizija bila je stvoriti sustav u kojem bi se ugovorne obveze automatski provodile bez potrebe za ljudskom intervencijom, čime bi se smanjila mogućnost pogrešaka i prijevara [9].

Pametni ugovori se temelje na logici "ako-onda" (*if-then*), što znači da određene radnje slijede unaprijed definirane uvjete. Na primjer, ako je uplaćen određeni iznos novca, tada se izdaje potvrda o vlasništvu. Ova jednostavna logika omogućuje pametnim ugovorima da automatiziraju širok raspon transakcija i procesa. Na primjer, u slučaju online kupovine, pametni ugovor može automatski osloboditi sredstva prodavatelju kada kupac potvrdi prijem proizvoda, čime se osigurava pravovremeno izvršenje transakcije.

Pametni ugovori se pohranjuju na lancu blokova, distribuiranoj knjizi koja bilježi sve transakcije na mreži. Lanac blokova je sigurna, transparentna i nepromjenjiva struktura podataka, što znači da se podaci pohranjeni na njemu ne mogu mijenjati bez konsenzusa mreže. Svaki blok na lancu blokova sadrži hash prethodnog bloka, vremensku oznaku i podatke o transakcijama, što osigurava integritet i kronološki slijed transakcija [8] [10].

Tehnologija lanca blokova omogućuje dva osnovna postupka: čitanje (eng. *read*) i dodavanje (eng. *append*). Čitanje omogućuje pregled sadržaja blokova, dok dodavanje omogućuje dodavanje novih blokova s transakcijama u knjigu. Ova svojstva osiguravaju transparentnost i sigurnost podataka pohranjenih na lancu blokova. Pametni ugovori koriste tehnologiju lanca blokova kako bi osigurali nepromjenjivost i transparentnost podataka. Nepromjenjivost znači da se podaci pohranjeni na lancu blokova ne mogu mijenjati bez

konsenzusa mreže, čime se osigurava integritet ugovora. Transparentnost omogućuje svim sudionicima u mreži da provjere sadržaj pametnih ugovora i prate njihov status izvršenja [8] [10].

Što se tiče ključnih karakteristika pametnih ugovora, bitno je istaknuti sljedeće [10]:

1. **Autonomija**: Pametni ugovori djeluju autonomno nakon implementacije. Jednom kada su pametni ugovori kreirani i pokrenuti na lancu blokova, ne zahtijevaju daljnju interakciju s pokretačima ili bilo kakvu ljudsku intervenciju. Ova karakteristika omogućuje ugovorima da samostalno izvršavaju svoje zadatke, smanjujući potrebu za administrativnim nadzorom i ljudskom pogreškom. Autonomija pametnih ugovora također znači da se sve transakcije odvijaju bez potrebe za posrednicima, što dodatno smanjuje troškove i povećava učinkovitost.
2. **Samodostatnost**: Pametni ugovori mogu samostalno upravljati resursima. Ovo uključuje sposobnost prikupljanja sredstava pružanjem usluga i njihovo trošenje po potrebi, na primjer, dobivanje računalne snage ili pohrane. Pametni ugovori mogu osigurati financijska sredstva putem različitih metoda, kao što su naknade za usluge ili mikro uplata. Samodostatnost omogućuje ugovorima da budu neovisni i operativno učinkoviti, što je posebno važno za aplikacije koje zahtijevaju kontinuirano funkcioniranje bez vanjskih intervencija.
3. **Decentraliziranost**: Pametni ugovori djeluju na decentraliziranim mrežama, što znači da ne postoje na jednom centraliziranom poslužitelju. Umjesto toga, distribuirani su i samostalno provedivi preko mrežnih čvorova. Ova decentralizacija osigurava otpornost pametnih ugovora na cenzuru i manipulaciju, jer ne postoji pojedinačna točka neuspjeha. Distribuirani karakter pametnih ugovora također povećava sigurnost i pouzdanost mreže, jer svaki čvor u mreži sadrži kopiju ugovora i može provjeriti njegovu valjanost.
4. **Nepromjenjivost**: Jedna od ključnih karakteristika pametnih ugovora je njihova nepromjenjivost. Nakon što je pametni ugovor implementiran na lanac blokova, njegov kod se ne može mijenjati. Ovo osigurava da se ugovorni uvjeti ne mogu retroaktivno mijenjati, čime se povećava povjerenje među ugovornim stranama. Nepromjenjivost također sprječava manipulaciju podacima i osigurava integritet ugovora. Ovo je posebno važno u poslovnim okruženjima gdje su transparentnost i povjerenje ključni za uspješno poslovanje.
5. **Transparentnost**: Pametni ugovori su javno dostupni na lancu blokova, što omogućuje svim stranama da provjere njihov sadržaj i status. Ova transparentnost povećava povjerenje među ugovornim stranama jer svaka strana može vidjeti kako će se ugovor

provesti. Javna priroda pametnih ugovora također omogućuje neovisnim promatračima da prate transakcije i osiguraju poštivanje pravila i propisa.

6. **Automatsko izvršenje**: Pametni ugovori automatski izvršavaju definirane radnje bez potrebe za posrednicima. Ovo smanjuje troškove i vrijeme potrebno za izvršenje ugovora, jer nema potrebe za ljudskom intervencijom ili administrativnim postupcima. Automatsko izvršenje također smanjuje rizik od pogrešaka i prijevara, jer se sve radnje provode automatski prema unaprijed definiranim pravilima. Ova karakteristika omogućuje brzo i učinkovito izvršenje transakcija, što je posebno važno u financijskim i trgovačkim aplikacijama.
7. **Sigurnost**: Pametni ugovori koriste kriptografske tehnike za osiguranje podataka i transakcija. Ovo osigurava visoku razinu sigurnosti i sprječava neovlašteni pristup ili manipulaciju podacima. Sigurnost pametnih ugovora temelji se na kriptografiji javnog ključa, koja osigurava da samo ovlaštene strane mogu pristupiti i izvršavati transakcije. Dodatno, decentralizirana priroda lanca blokova osigurava da su svi podaci sigurno pohranjeni i da se ne mogu mijenjati bez konsenzusa mreže.

Ove karakteristike čine pametne ugovore atraktivnim rješenjem za mnoge industrije i primjene, omogućujući sigurnu, efikasnu i transparentnu provedbu ugovornih obveza. U nastavku, da bi se pametni ugovori lakše shvatili, su stavljeni par primjera iz stvarnog života.

Istraživanja o pametnim ugovorima pokazala su da oni imaju značajan potencijal za transformaciju poslovnih procesa u različitim industrijama. U razdoblju od 2012. do 2022. godine, obavljeno je opsežno istraživanje koje je obuhvatilo 252 članka s ključnim pojmovima poput "lanac blokova" i "pametni ugovori". Ova istraživanja su pokazala da pametni ugovori imaju potencijal za smanjenje troškova, povećanje učinkovitosti i poboljšanje transparentnosti u poslovanju. Međutim, istraživanje je također identificiralo nekoliko ključnih izazova, uključujući pravne prepreke koje su gore navedene, tehničke složenosti i potrebu za pouzdanim izvorima podataka. Istraživači su istaknuli da je potrebno razviti nove metodologije i najbolje prakse za razvoj i testiranje pametnih ugovora kako bi se osigurala njihova sigurnost i učinkovitost [11].

4.2.1. Primjeri iz stvarnog života

Da bi se princip rada pametnih ugovora lakše razumio u nastavku ću dati primjer kako stvari iz stvarnog života funkcioniraju na slični princip kao što funkcioniraju pametni ugovori.

Jedan od najjednostavnijih primjera pametnih ugovora je automatizirani prodajni stroj. Kada ubacite novac u stroj i odaberete proizvod, stroj automatski izdaje proizvod ako je iznos točan. Ovaj proces ne zahtijeva ljudsku intervenciju jer je sve unaprijed programirano. Stroj provjerava iznos, potvrđuje izbor i izdaje proizvod. Slično, pametni ugovor automatski provodi ugovorne obveze kada su ispunjeni uvjeti definirani u kodu.

Sljedeći primjer bi bio platforma za online prodaju. Zamislite platformu za online trgovinu koja koristi pametne ugovore za transakcije. Kupac uplaćuje sredstva na escrow račun putem pametnog ugovora. Prodavatelj zatim šalje proizvod, a kada kupac potvrdi prijem, pametni ugovor automatski oslobađa sredstva prodavatelju. Ako se pojavi spor, pametni ugovor može uključiti treću stranu koja će pružiti relevantne informacije kako bi se riješio spor. Primjena pametnih ugovora u online trgovini može značajno povećati sigurnost i povjerenje među korisnicima. Pametni ugovor osigurava da se sredstva automatski oslobađaju prodavatelju tek nakon što kupac potvrdi prijem proizvoda. Ovo smanjuje rizik od prijevara i osigurava pravovremeno izvršenje transakcija.

Drugi primjer bi bio u industriji osiguranja, pametni ugovori mogu automatizirati isplatu osiguravajućih naknada. Na primjer, pametni ugovor za osiguranje usjeva može biti programiran tako da automatski isplati naknadu poljoprivredniku ako meteorološki podaci pokazuju da je došlo do suše ili poplave.

4.3. Pravni izazovi i regulacija

Jedan od najvećih izazova pametnih ugovora je njihova pravna regulacija. Većina pravnih sustava u svijetu još uvijek nije prilagodila svoje zakonodavstvo kako bi u potpunosti integrirala pametne ugovore. Dodatno, mnogi pravni koncepti teško se mogu izraziti u kodu zbog njihove složene prirode. Budući da se pametni ugovori izvršavaju na globalnoj mreži lanca blokova, može biti teško odrediti koja pravila i zakoni se primjenjuju na određeni ugovor. Ova pravna nesigurnost može predstavljati prepreku za širu primjenu pametnih ugovora u reguliranim industrijama.

U Hrvatskoj, od 2024. godine, došlo je do značajnih regulatornih promjena u vezi s digitalnim tržištem, uključujući regulaciju kriptovaluta. Međutim, specifični zakoni koji se izravno bave pametnim ugovorima nisu izričito spomenuti, što znači da se zakonski okvir za pametne ugovore još uvijek razvija ili izravno ne postoji. Globalno gledano, nekoliko zemalja je počelo pružati pravnu osnovu za pametne ugovore. Italija i različite američke savezne države donijele su zakone koji priznaju valjanost i izvršnost pametnih ugovora. Talijansko zakonodavstvo definira pametne ugovore kao vrstu elektroničkog zapisa koji ispunjava zakonske zahtjeve za pisane ugovore, pod uvjetom da su ispunjeni određeni uvjeti. Slično

tome, američke države poput Arizone izmijenile su svoje Zakone o elektroničkim transakcijama kako bi uključile pametne ugovore, osiguravajući im isti pravni status kao tradicionalnim ugovorima [12].

4.3.1. Oracle problem

Pametni ugovori često ovise o vanjskim podacima kako bi pravilno funkcionirali. *Oracles* su posrednici koji omogućuju povezivanje mreža lanaca blokova s vanjskim svijetom pružajući podatke i usluge koji su potrebni za izvršenje ugovora. *Oracles* omogućuju lancu blokova pristup vanjskim informacijama koje nisu izvorno dostupne na lancu blokova [8] [11].

Na primjer, za pametni ugovor o trgovanju dionicama, *oracle* može dostaviti podatke o trenutnim cijenama dionica koje pokreću izvršenje trgovačkih naloga kada cijena dosegne određeni prag. Pouzdanost i sigurnost *oracla* je kritična, jer nepravilni ili zlonamjerni podaci mogu ugroziti izvršenje pametnog ugovora [8] [11].

Oracles mreže omogućuju pametnim ugovorima pristup postojećim izvorima podataka, naslijeđenim sustavima i naprednim proračunima. Decentralizirane oracle mreže (eng. *Decentralised Oracle network,* skraćeno DON) omogućuju stvaranje hibridnih pametnih ugovora, gdje se *onchain* kod i *offchain* infrastruktura kombiniraju kako bi podržali napredne decentralizirane aplikacije (skraćeno dApps) koje reagiraju na stvarne događaje i operiraju s tradicionalnim sustavima. Na primjer, Chainlink oracle mreža omogućila je stvaranje DeFi prostora i postala industrijski standard oracle rješenja za cijeli Web3. Chainlink sada surađuje s velikim financijskim institucijama, omogućujući napredne, sigurne i interoperabilne blockchain aplikacije [8] [11].

Jedan od ključnih izazova vezanih uz *oracles* je osiguranje točnosti i pouzdanosti podataka koje dostavljaju. Moguće rješenje je korištenje više *oracla* i konsenzusnog mehanizma za verifikaciju podataka, što bi povećalo pouzdanost podataka, ali i složenost i troškove sustava.

Pametni ugovori predstavljaju značajan tehnološki napredak te njihova primjena može drastično smanjiti troškove i vrijeme transakcija, povećati transparentnost i eliminirati potrebu za posrednicima. Međutim, pravna regulacija i pouzdanost oracla ostaju ključni izazovi. Smatram da će daljnji razvoj tehnologije i standardizacija protokola omogućiti još širu primjenu pametnih ugovora, čime će se dodatno povećati njihova učinkovitost i sigurnost u poslovnim procesima. Budućnost pametnih ugovora ovisi o njihovoj sposobnosti da se prilagode pravnim i tehničkim izazovima te o suradnji između tehnoloških inovatora i pravnih stručnjaka kako bi se osigurala njihova učinkovita i sigurna primjena.

5. Solidity programski jezik

Solidity je programski jezik razvijen za pisanje pametnih ugovora koji se izvršavaju na Ethereum platformi. Riječ je o objektno orijentirani jeziku više razine, inspiriran je jezicima poput JavaScripta, Pythona i C++, čineći ga pristupačnim za programere s iskustvom u ovim jezicima. Njegova glavna svrha je omogućiti programerima stvaranje složenih, sigurnih i efikasnih pametnih ugovora koji se mogu automatizirati i izvršavati unutar decentralizirane blockchain mreže [13].

Solidity se pokreće na Ethereum Virtual Machine (EVM), što znači da je dizajniran da bude kompajliran u bytecode koji EVM može izvršavati. Kao statički tipiziran jezik, svaka varijabla mora imati definiran tip, čime se povećava sigurnost i smanjuje mogućnost grešaka tijekom izvršavanja. Osim toga, Solidity podržava nasljeđivanje, biblioteke i složene korisnički definirane tipove podataka.

Ključne značajke Soliditya su sljedeće [13]:

1. **Statička tipizacija**: Tipovi podataka se moraju eksplicitno definirati prilikom deklaracije varijabli.
2. **Podrška za nasljeđivanje:** Pametni ugovori mogu nasljeđivati druge ugovore, omogućujući ponovno korištenje koda.
3. **Biblioteke (eng. *Libaries*):** Omogućuju dijeljenje zajedničkih funkcija i smanjenje dupliciranja koda. Riječ je o posebnoj vrste ugovora koji sadrže zajedničke funkcije koje se mogu ponovno koristiti u drugim ugovorima. Biblioteke omogućuju modularnost i dijeljenje koda. Biblioteke se ne mogu implementirati samostalno na lancu blokova, već se koriste unutar drugih ugovora. Biblioteke ne mogu imati stanje (eng. *state variables*) niti mogu držati Ether. Svi podaci koje biblioteke koriste moraju biti proslijeđeni kao argumenti funkcijama. Biblioteke ne podržavaju nasljeđivanje te unutar njih se mogu definirati unutarnje (eng. *internal*) funkcije koje se mogu pozivati samo unutar ugovora u kojem su definirane.
4. **Strukture i nabrajanja:** Omogućuju definiranje složenih tipova podataka koji mogu grupirati više vrijednosti.

5.1. Struktura i sintaksa

Solidity koristi sintaksu sličnu mnogim programskim jezicima, što olakšava njegovo usvajanje. U ovom poglavlju ću navesti osnovne elemente sintakse i struktura koje se koriste

pri pisanju pametnih ugovora. Da bi se kreirao ugovor, potrebno je koristiti ključnu riječ `contract` te je sintaksa sljedeća:

```
contract nazivUgovora  {
}
```

U Solidity-u, varijable se mogu deklarirati unutar ugovora. Tipovi podataka koje Solidity podržava su sljedeći:

Tablica 1: Tipovi podataka u Solidity-u

Tip Podataka	Opis
bool	Logički tip podataka, može imati vrijednosti true ili false.
string	Niz znakova.
uint	Neznamenkasti cijeli brojevi. Mogu biti od 8 do 256 bitova u koracima od 8 (npr. uint8, uint16, uint32, ... uint256). uint je sinonim za uint256.
int	Znamenkasti cijeli brojevi. Mogu biti od 8 do 256 bitova u koracima od 8 (npr. int8, int16, int32, ... int256). int je sinonim za int256.
fixed	Fiksni decimalni brojevi (trenutno nisu potpuno podržani).
ufixed	Neznamenkasti fiksni decimalni brojevi (trenutno nisu potpuno podržani).
address	Ethereum adresa veličine 20 bajtova. address payable je adresa koja može primati Ether.
array	Nizovi spremaju elemente istog tipa u jednu kolekciju. Mogu biti fiksne i dinamičke duljine.
mapping	Mapa koja predstavlja kolekciju u obliku ključ-vrijednost. Ključevi mogu biti tipa uint, bool ili address, a vrijednosti mogu biti bilo kojeg tipa.
enum	Enumeracija koja omogućuje varijabli da poprimi samo jednu od unaprijed definiranih vrijednosti.
struct	Složeni tip podataka koji korisnik sam definira i spaja više različitih tipova u jedan.

Nadalje, funkcije su ključni dijelovi pametnih ugovora. One definiraju akcije koje ugovor može poduzeti. Svaka funkcija započinje s ključnom riječi `function`, slijedi naziv funkcije, lista parametara, definicija vidljivosti, te povratne vrijednosti. Oblik funkcije je sljedeći:

```
function ime(parametri) vidljivost returns(return_value) {}
```

Vidljivost funkcija određuje tko može pozvati funkciju:

- `public`: Funkcija je dostupna svima.
- `private`: Funkcija je dostupna samo unutar ugovora.
- `internal`: Funkcija je dostupna unutar ugovora i ugovora koji nasljeđuju taj ugovor.
- `external`: Funkcija je dostupna samo vanjskim ugovorima i transakcijama.

Modifikatori funkcija omogućuju provjeru uvjeta prije ili nakon izvršenja funkcije. Najčešće se koriste za provjeru prava pristupa ili stanja ugovora. Modifikator je kao predfunkcija koja se izvršava prije glavne funkcije i može kontrolirati hoće li se glavna funkcija izvršiti ili ne. U nastavku slijedi modifikatora koji provjerava je li pozivatelj funkcije vlasnik ugovora:

```
contract MojUgovor {
    // Varijabla koja pohranjuje adresu vlasnika ugovora
    address public owner;
    // Varijabla koja pohranjuje broj
    uint public mojBroj;

    // Konstruktor koji postavlja vlasnika ugovora kao adresu koja je
implementirala ugovor
    constructor() {
        owner = msg.sender;
    }

    // Modifikator koji provjerava je li pozivatelj funkcije vlasnik
ugovora
    modifier samoVlasnik() {
        require(msg.sender == owner, "Niste vlasnik ugovora");
        // _; označava mjesto gdje će se nastaviti izvršenje funkcije
koja koristi ovaj modifikator
        _;
    }

    // Funkcija koja postavlja vrijednost varijable 'mojBroj', ali
samo ako je pozivatelj vlasnik ugovora
    function postaviMojBroj(uint _vrijednost) public samoVlasnik {
        mojBroj = _vrijednost;
    }
}
```

Biblioteka (eng. *library*) se definira koristeći ključnu riječ `library`. Unutar biblioteke se definiraju funkcije koje se mogu koristiti u drugim ugovorima. U nastavku slijedi primjer kako napraviti library.

```
library Matematika {
    function zbroji(uint a, uint b) internal pure returns (uint) {
        return a + b;
    }

    function oduzmi(uint a, uint b) internal pure returns (uint) {
        require(b <= a, "Provjera");
        return a - b;
    }
```

```
}
```

U ovom primjeru, biblioteka `Matematika` sadrži dvije funkcije: zbroji koja zbraja dva broja i oduzmi koja oduzima jedan broj od drugog uz provjeru. Biblioteka se može koristiti u ugovoru pomoću ključne riječi **`using`**.

```
import "./Matematika.sol";

contract MojUgovor {
    using Matematika for uint;

    function primjerKoristenja(uint a, uint b) public pure returns
(uint) {
        uint sum = a.add(b);
        uint difference = a.sub(b);
        return sum + difference;
    }
}
```

U ovom primjeru, ugovor `MojUgovor` koristi biblioteku `Matematika` za izvođenje matematičkih operacija. Ključna riječ **`using`** `Matematika` **`for`** **`uint`** omogućuje pozivanje funkcija iz biblioteke kao metoda za tip **`uint`**.

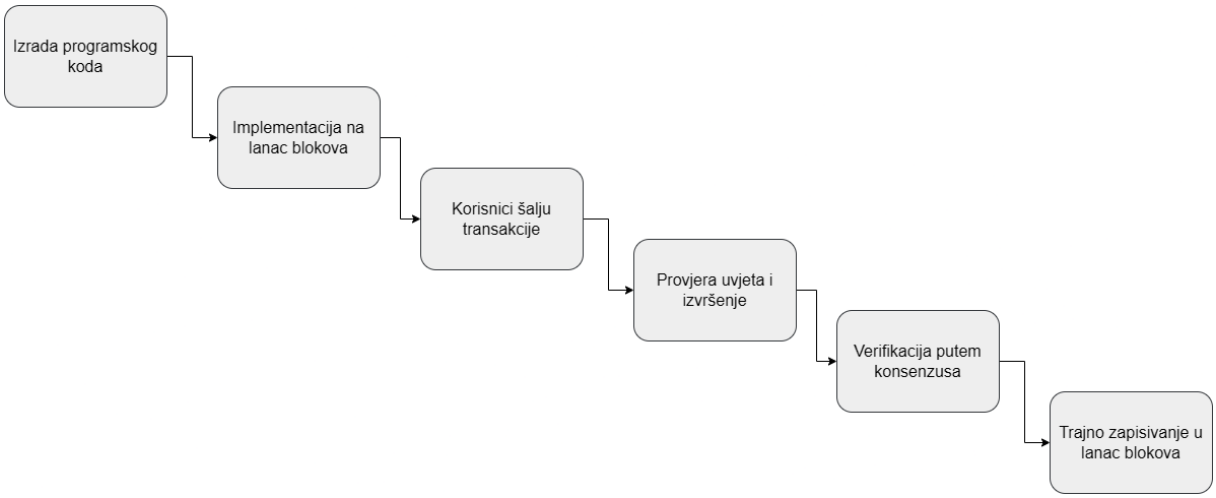
Kao što je već prije bilo spomenuto, Solidity podržava nasljeđivanje ugovora, što omogućava jednom ugovoru da nasljeđuje funkcionalnost drugog ugovora. To omogućuje ponovno korištenje koda i modularnost. U nastavku je dan primjer kako se to izrađuje:

```
contract NapredniUgovor is OsnovniUgovor {
    // Mapa za praćenje depozita korisnika
    mapping(address => uint) public depoziti;

    // Funkcija za primanje depozita
    function primiDepozit() public payable {
        require(msg.value > 0, "Depozit mora biti veći od nule");
        depoziti[msg.sender] += msg.value;
    }

    // Funkcija za povlačenje vlastitih depozita
    function povuciDepozit(uint _iznos) public {
        require(depoziti[msg.sender] >= _iznos, "Nedovoljno
sredstava");
        depoziti[msg.sender] -= _iznos;
        payable(msg.sender).transfer(_iznos);
    }

    // Funkcija za prikaz ukupnog depozita korisnika
    function ukupniDepozit() public view returns (uint) {
        return depoziti[msg.sender];
    }
}

// Glavni ugovor koji demonstrira nasljeđivanje i korištenje
funkcionalnosti
contract GlavniUgovor {
    // Inicijalizacija ugovora 'NapredniUgovor'
    NapredniUgovor public napredniUgovor;

    // Konstruktor koji stvara novu instancu ugovora 'NapredniUgovor'
```

```
    constructor() {
        napredniUgovor = new NapredniUgovor();
    }

    // Funkcija za primanje depozita putem glavnog ugovora
    function primiDepozit() public payable {
        napredniUgovor.primiDepozit{value: msg.value}();
    }

    // Funkcija za prikaz balansa ugovora
    function provjeriBalans() public view returns (uint) {
        return napredniUgovor.provjeriBalans();
    }

    // Funkcija za povlačenje vlastitih depozita putem glavnog
ugovora
    function povuciDepozit(uint _iznos) public {
        napredniUgovor.povuciDepozit(_iznos);
    }

    // Funkcija za prikaz ukupnog depozita korisnika putem glavnog
ugovora
    function ukupniDepozit() public view returns (uint) {
        return napredniUgovor.ukupniDepozit();
    }
}
```

5.2. Razvoj pametnih ugovora u Solidity-u

Kako bismo bolje razumjeli kako se razvijaju pametni ugovori, kreiran je dijagram. Dijagram ispod prikazuje ključne korake u procesu izrade i izvršenja pametnog ugovora. Svaki korak je ključan za osiguravanje funkcionalnosti, sigurnosti i transparentnosti pametnih ugovora. Ovdje su objašnjeni svi elementi s dijagrama:



Slika 1. Proces izrade i izvrašavanja pametnog ugovora

1. **Izrada programskog koda**: U ovom koraku, programer koristi programski jezik poput Solidity da napiše kod pametnog ugovora. Ovaj kod sadrži sve uvjete i pravila koja će se automatski izvršavati kada su određeni uvjeti ispunjeni. Ovo je najbitniji korak zbog toga što kvaliteta i sigurnost koda direktno utječu na funkcionalnost i pouzdanost pametnog ugovora. Svaka funkcija unutar ugovora započinje s ključnom riječi function iza koje slijedi naziv funkcije i lista parametara. Na primjer:

```
function transfer(address recipient, uint amount) public {
    // Kod funkcije }
```

2. **Implementacija na lanac blokova:** Nakon što je kod napisan, on se implementira na blockchain mreži, poput Ethereum lanca blokova. Ovo znači da je pametni ugovor sada dostupan na decentraliziranoj mreži gdje može biti izvršen.
3. **Korisnici šalju transakcije**: Korisnici ili druge aplikacije interagiraju s pametnim ugovorom slanjem transakcija koje aktiviraju određene funkcije unutar ugovora.
4. **Provjera uvjeta i izvršenje:** Pametni ugovor automatski provjerava postavljene uvjete. Ako su svi uvjeti ispunjeni, ugovor izvršava definirane radnje, kao što je prijenos sredstava s jednog računa na drugi.
5. **Verifikacija putem konsenzusa:** Transakcije i rezultati izvršenja pametnog ugovora moraju biti verificirani putem mrežnog konsenzusa. U mrežama poput Ethereuma, to uključuje proces rudarenja ili *stakinga* gdje čvorovi potvrđuju valjanost transakcija.
6. **Trajno zapisivanje u lanac blokova:** Nakon što su transakcije verificirane, one se trajno zapisuju u lanac blokova. Ovaj zapis je nepromjenjiv i može se pregledati u bilo kojem trenutku, što osigurava transparentnost.

Da bi se to još bolje razumjelo, evo kratkog primjera koda za razvoj pametnog ugovora koji prati korake iznad. Ovaj primjer prikazuje osnovne korake u razvoju pametnog ugovora, uključujući definiranje funkcija, postavljanje uvjeta za izvršavanje i omogućavanje prijenosa sredstava. Naravno, riječ je primjeru, konkretni programski kod i razvoj pametnog ugovora će biti prikazan u fazi izrade.

```
contract MojUgovor {
    // Državna varijabla za pohranu vlasnika ugovora
    address public vlasnik;

    // Konstruktor koji postavlja vlasnika ugovora
    constructor() {
        vlasnik = msg.sender;
    }

    // Funkcija za prijenos sredstava na zadanu adresu
    function transfer(address recipient, uint amount) public {
```

```
        require(msg.sender == vlasnik, "Samo vlasnik može izvršiti
prijenos");
        payable(recipient).transfer(amount);
    }

    // Funkcija za primanje depozita
    function primiDepozit() public payable {}
}
```

6. React

Odabrala sam React za praktični dio svog front-end razvoja zbog njegove fleksibilnosti, snažnog ekosustava, i značajnih prednosti koje nudi u izgradnji dinamičnih i interaktivnih korisničkih sučelja. React je razvijen od strane Facebooka te omogućuje stvaranje ponovljivih elemenata korisničkog sučelja (eng. *User Interface*, UI) kroz komponentno baziranu arhitekturu, što pojednostavljuje upravljanje i održavanje kompleksnih web aplikacija [14].

Željela sam koristiti React zbog nekoliko ključnih značajki koji ima [14]:

- komponentno bazirana arhitektura
- virtualni DOM (eng. *Document Object Model*)
- jednosmjerna propusnost podataka
- React Hooks
- široka podrška zajednice

Svaka značajka će biti detaljnije objšanjena u nastavku. Komponentno bazirana arhitektura znači da React omogućava razvoj komponenti koje su ponovljive i neovisne, što olakšava razvoj složenih aplikacija. Svaka komponenta može biti testirana i razvijana zasebno, što ubrzava razvojni proces i poboljšava održavanje koda.

Virtualni DOM (eng. *Document Object Model*): React koristi virtualni DOM za optimizaciju performansi aplikacija. Virtualni DOM je predstavljen kao lagana kopija stvarnog DOM-a. Kada se promjene dogode u aplikaciji, umjesto direktne manipulacije stvarnog DOM-a, prvo se ažurira virtualni DOM. Nakon toga, virtualni DOM uspoređuje svoje prethodno stanje s novim stanjem (proces poznat kao "diffing") i izračunava minimalni skup promjena potrebnih za ažuriranje stvarnog DOM-a. Ovaj pristup minimizira nepotrebno ponovno renderiranje i poboljšava brzinu aplikacije.

Nadalje, jednosmjerna propusnost podataka znači da podaci teku u jednom smjeru: od roditeljske komponente prema djetetovim komponentama. U Reactu podaci se prenose pomoću *props* (svojstava), koje roditeljska komponenta prosljeđuje svojim djetetovim komponentama. Evo primjer:

```
import React, { useState } from 'react';

function RoditeljskaKomponenta() {
  const [podaci, setPodaci] = useState("Pozdrav");

  return <DijeteKomponenta podaci={podaci} />;
}
```

```
function DijeteKomponenta(props) {
  return <p>{props.podaci}</p>;
}
export default RoditeljskaKomponenta;
```

U ovom primjeru, `RoditeljskaKomponenta` koristi stanje (`useState`) za spremanje podatka `Pozdrav` koji se zatim prosljeđuje `DjeteKomponenta` putem propova.

Zatim**,** React Hooks su funkcije koje omogućuju korištenje React značajki kao što su stanje (*state*) i životni ciklus komponenti (*lifecycle methods*) unutar funkcionalnih komponenti. Prije uvođenja hookova ove značajke su bile dostupne samo u klasnim komponentama. Uvođenje React Hookova, poput `useState` i `useEffect` omogućuje se korištenje stanja i drugih React značajki unutar funkcionalnih komponenti. Hooks pojednostavljuju kod i čine ga čitljivijim i lakšim za održavanje. Da bi se lakše razumijelo, imamo sljedeći primjer:

```
import React, { useState, useEffect } from 'react';

function Primjer() {
  const [broj, postaviBroj] = useState(0);

  useEffect(() => {
    console.log(`Kliknuli ste ${broj} puta`);
  }, [broj]);

  return (
    <div>
      <p>Kliknuli ste {broj} puta</p>
      <button onClick={() => postaviBroj(broj + 1)}>
        Klikni me
      </button>
    </div>
  );
}
export default Primjer;
```

U ovom primjeru, `useState` se koristi za praćenje broja klikova, a useEffect za ispisivanje poruke u konzolu svaki put kada se broj promijeni.

I posljednji bitan razlog, široka podrška zajednice. React ima ogromnu zajednicu i bogat ekosustav alata i knjižnica. To uključuje alate kao što su Redux za upravljanje stanjem i React Router za navigaciju, što dodatno olakšava razvoj složenih aplikacija.

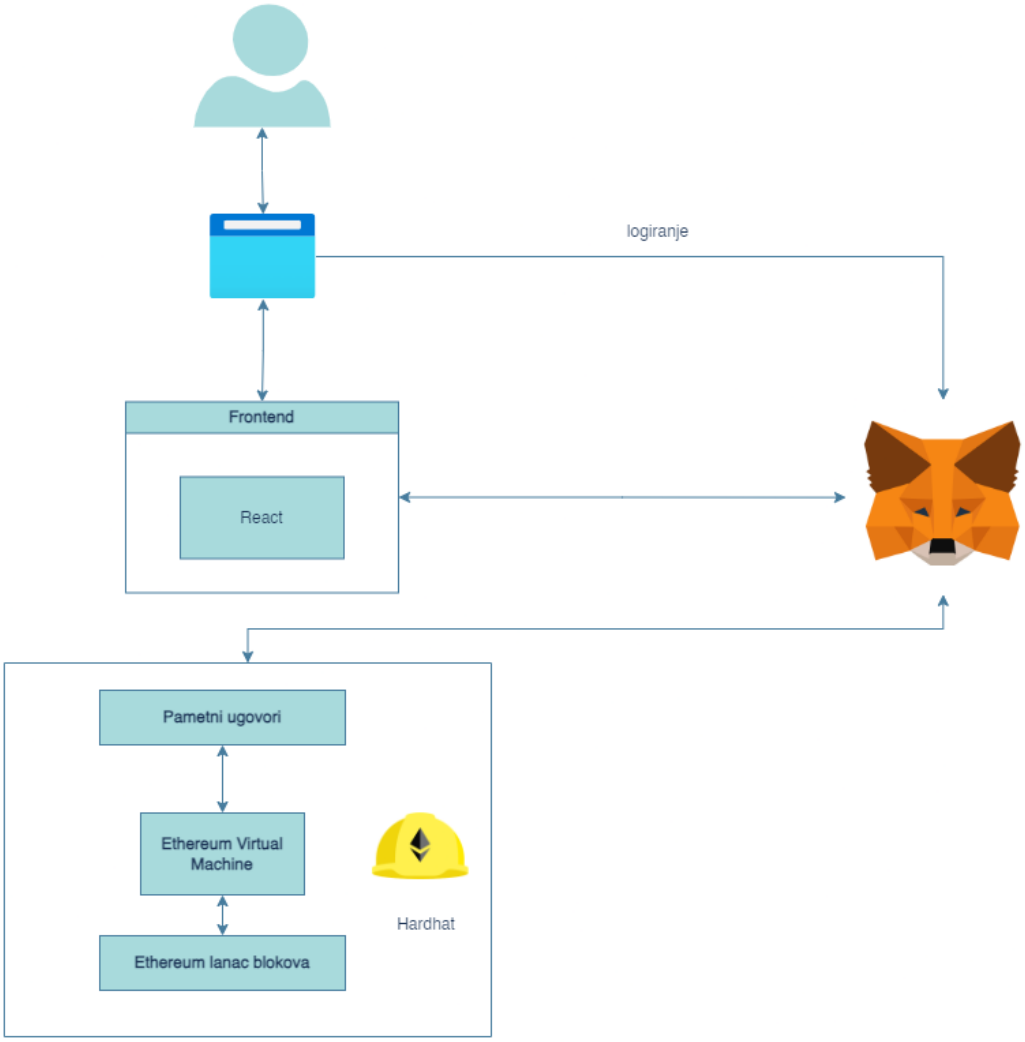
React je moćan alat za izgradnju modernih web aplikacija zbog svoje fleksibilnosti, performansi i bogatog ekosustava. Ove značajke ga čine idealnim izborom za razvoj dinamičnih i responzivnih korisničkih sučelja. Kroz učenje React-a, željela sam poboljšati svoje vještine i sposobnosti u front-end razvoju te bolje razumjeti suvremene pristupe u razvoju web aplikacija.

7. Razvoj decentralizirane aplikacije

U nastavku će biti objašnjen praktični dio rada, razvoj aplikacije.

7.1. Pregled arhitekture aplikacije

Arhitektura aplikacije je takva da u aplikaciji ne postoji centralizirana baza podataka koja pohranjuje stanje aplikacije. To znači da za razliku od centraliziranih aplikacija, klijentska strana aplikacije ne komunicira s bazom podataka, već izravno s lancem blokova. Konkretno, arhitektura izrađene aplikacije je sljedeća:



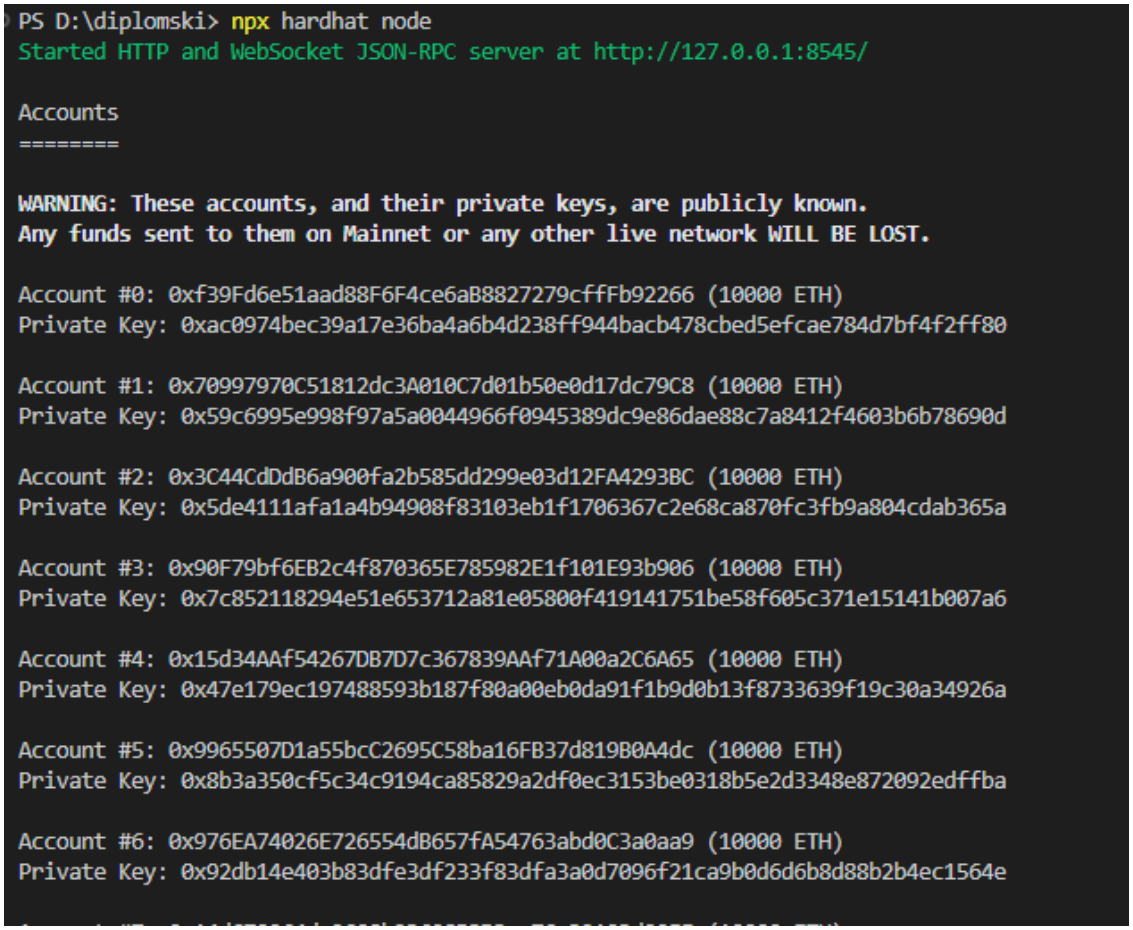
Slika 2. Arhitektura aplikacije

Korisnik pristupa aplikaciji putem preglednika na localhostu. Preglednik učitava frontend aplikaciju izrađenu u Reactu. Odmah prilikom pristupa aplikaciji, korisnik se mora

ulogirati u Metamask novčanik kako bi mogao slati transakcije. Prilikom postavljanja novčanika u Metamasku, potrebno je postaviti da se da se Metamask račun povezuje s Hardhat lokalnom mrežom. Kada korisnik želi izvršiti transakciju ili interakciju s pametnim ugovorima, Metamask se koristi za autentifikaciju i potpisivanje transakcija. Nadalje, kada se pametni ugovor deploya, Hardhat šalje transakciju koja uključuje ugovor na lanac blokova, u ovom slučaju Hardhat lokalnu mrežu. Pametni ugovor je deployan lokalno na Hardhat mrežu gdje se sva logika izvršava na Ethereum Virtual Machine (EVM). U sljedećem poglavlju je detaljnije objašnjeno kako se to izvršava.

7.2. Demonstracija procesa kupnje i plaćanja Ethereumom

U nastavku će biti demonstrirano kako se aplikacija koristi i objašnjeni najbitniji dijelovi programskog koda. Da bi se prvo aplikacija koristila, potrebno je lokalno uključiti Hardhat mrežu. Hardhat ujedno služi kao alat za deployment pametnih ugovora i odmah nudi mogućnost pokretanja Ethereum mreže. Nažalost, za svoju mrežu ne koristi nikakav GUI pa je zbog toga sva interakcija unutar mreže vidljiva u terminalu. Za deployment se prije koristio Truffle koji se povezao s Ganache mrežom, nažalost trenutno su oba alata zastarjela i ne mogu se više koristiti. U terminalu je potrebno napisati `npx hardhat node` te se potom pokreće lokalna Ethereum mreža kao što je prikazano na slici ispod.



Slika 3. Hardhat lokalna Ethereum mreža

Nakon što je lokalna mreža pokrenuta potrebno je deployati pametni ugovor. To se može napraviti na više način, putem konzole ili koristeći Remix IDE. Ako se želi ugovor deployati putem konzole, prvo je potrebno izvršiti naredbu `npx hardhat compile` koja kompajlira ugovor. Nakon toga se treba izvršiti naredba `npx hardhat ignition deploy ./ignition/modules/SimpleMarketplace.js` .Ta skripta izgleda ovako:

```
const { buildModule } = require("@nomicfoundation/hardhat-
ignition/modules");

const SimpleMarketplaceModule = buildModule("SimpleMarketplaceModule", (m)
=> {
  const marketplace = m.contract("SimpleMarketplace");

  return { marketplace };
});

module.exports = SimpleMarketplaceModule;
```

Unutar skripte se koristi funkcija za buildanje pametnih ugovora. Metodda m.contract poziva ugovor koji se zove SimpleMarketplace te se to sprema u konstantu nazvanu marketplace. Funkcija vraća objekt koji sadrži instancu pametnog ugovora. Da bi se ista stvar

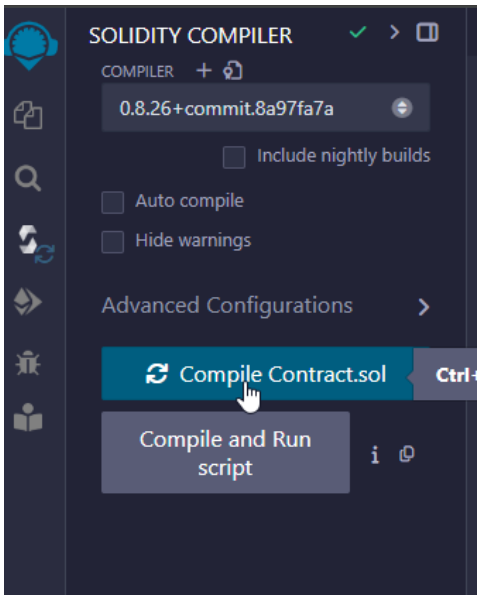
napravila u Remix IDE, potrebno je dodati .sol datoteku koja sadrži logiku ugovora. Programski kod za ugovor je sljedeći:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract SimpleMarketplace {
    struct Item {
        uint id;
        uint price;
        address seller;
    }

    mapping(uint => Item) public items;
    uint public itemCount;

    event ItemPurchased(uint indexed id, address indexed buyer, uint256
price);

    constructor() {
        // Konstruktor koji hardcodira artikle iz items.json
        listItem(1, 2 ether);          // Dell XPS 13
        listItem(2, 4 ether);          // Samsung Galaxy S25
        listItem(3, 7 ether);          // Apple MacBook Pro 16
        listItem(4, 5 ether);          // Google Pixel 6
        listItem(925927, 9 ether);     // Dyson V11
        listItem(847859, 8 ether);     // HP Spectre x360
        listItem(848148, 4 ether);     // Philips Hue
        listItem(848633, 900000 ether);    // Nest Thermostat
    }

    function listItem(uint id, uint price) public {
        items[id] = Item({
            id: id,
            price: price,
            seller: msg.sender
        });
        itemCount++;
    }

    function purchaseItem(uint itemId) external payable {
        Item storage item = items[itemId];
        require(item.seller != address(0), "Item does not exist");
        require(item.price == msg.value, "Incorrect price sent");

        // Transfer sredstava prodavatelju
        (bool success, ) = item.seller.call{value: msg.value}("");
        require(success, "Transfer to seller failed.");

        emit ItemPurchased(itemId, msg.sender, msg.value);
    }
}
```

Ugovor je kreiran tako da su se hardkodirali podaci vezani uz proizvode. Proizvod ima informacije o id, cijeni i tko je prodavač. Prvo se napravi struktura Item te se zatim koristi mapping kako bi se povezao id s podacima o tom artiklu. Dodan je itemCount brojač koji prati ukupan broj artikla na tržištu. Event itemPruchase se emitira kada kupac kupi artikl i sadrži najbitnije informacije. U konstruktoru se prilikom deployanja ugovora inicijalno dodaje nekoliko

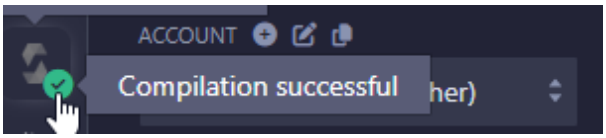
artikla na tržište čije informacije se poklapaju s informacijama koje frontend dobiva o ugovoru iz specifične .JSON datoteke. Dodana je funkcija listItem koja omogućuje dodavanje novih artikla na tržište. Najbitnija funkcija je purchaseItem koja funkcionira tako da se prvo pronađe artikl prema njegovom id-u. Potom se provjerava postoji li artikl s navedenom vrijednosti te je li poslana točna cijena za svaki proizvod. Ako su svi uvjeti zadovoljeni, transferiraju se sredstva prodavatelju koristeći call metodu.

Kad se .sol datoteka doda na Remix, potrebno je kliknuti gumb „Compile and Run script“.



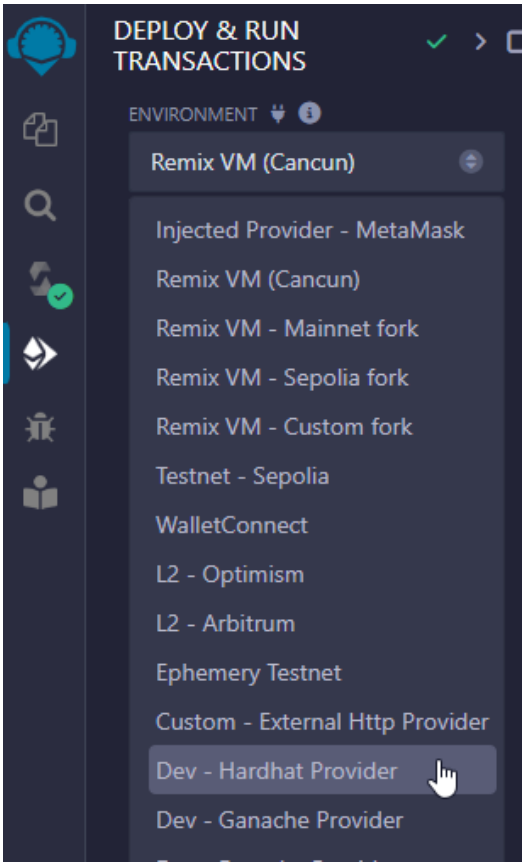
Slika 4. Remix IDE gumb za kompajliranje

Kad se .sol datoteka doda na Remix, potrebno je kliknuti gumb „Compile and Run script“. Kada se ugovor uspješno kompajlira, izbacuje se poruka o tome te se može proći na Deploy.



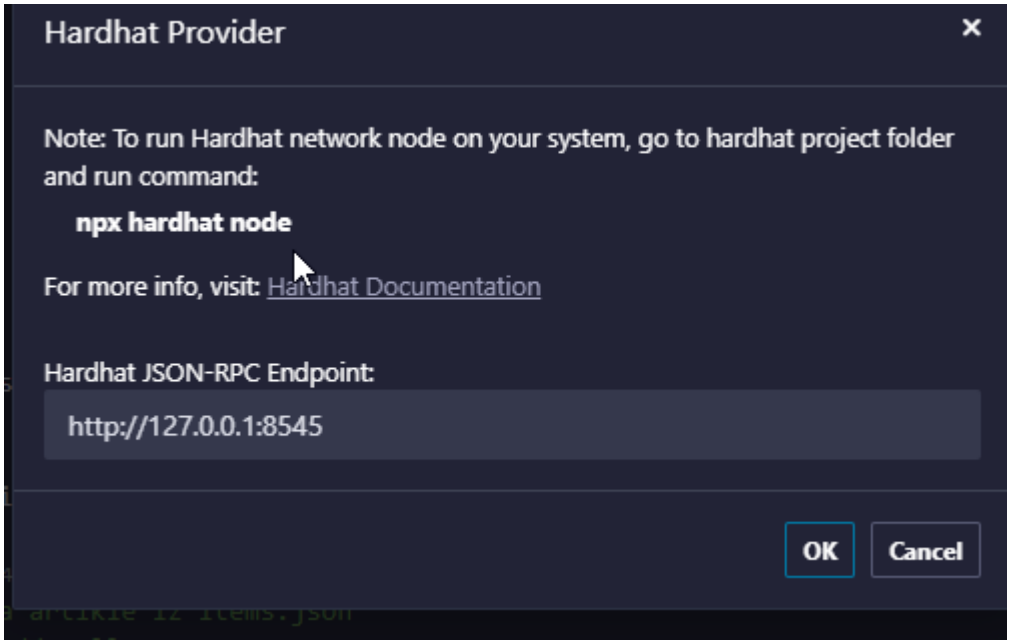
Slika 5. Poruka o uspješnom kompajliranju ugovora

Nadalje, prilikom deploymanta potrebno je kliknuti na Hardhat mrežu koja je lokalno već pokrenuta. Kako se to radi može se vidjeti na slici ispod.



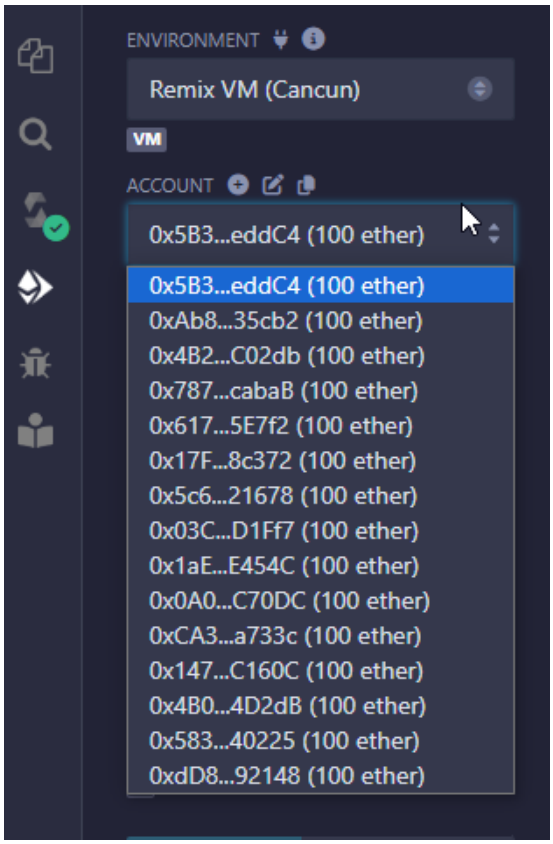
Slika 6. Odabir mreže gdje će se ugovor deployati

Nakon što se klikne gumb Deploy, REMIX nas pita potvrdno je li JSON-RPC dobro postavljen.



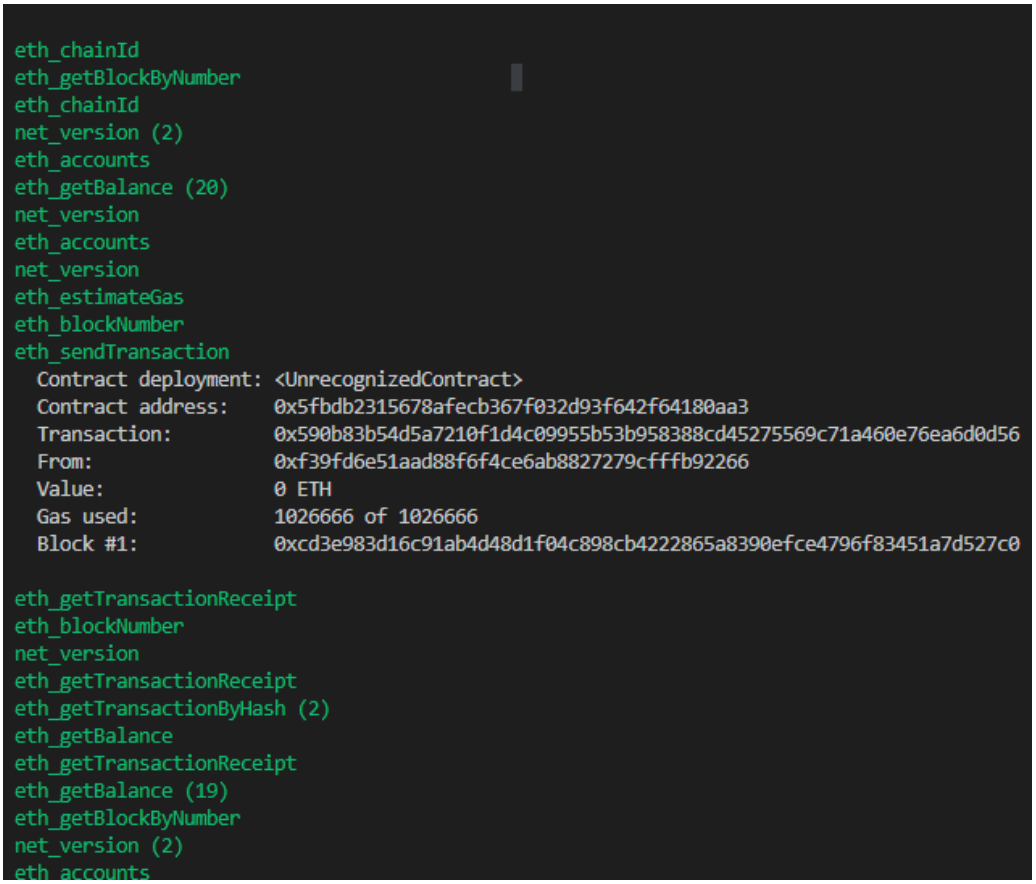
Slika 7. Remix pita je li mreža postavljena na dobobrom RPC-u

Ta informacija će nam još biti bitna kada ćemo ručno dodavati Hardhat mrežu u Metamask, zasad je dovoljno kliknuti samo ok. Prilikom deploymanta unutar Remixa se može izabrati s koje adrese želimo deployati ugovor. To su 10 adresa koje Hardhat mreža nudi prilikom pokretanja mreže koje služe za testiranje.



Slika 8. Adrese dostupne na Hardhat mreži

Kada se ugovor deploya, potrebno se vratiti u terminal gdje nam je Hardhat mreža pokrenuta da vidimo je li se ugovor uspješno deployao. U ovom slučaju možemo vidjeti na slici ispod da je:



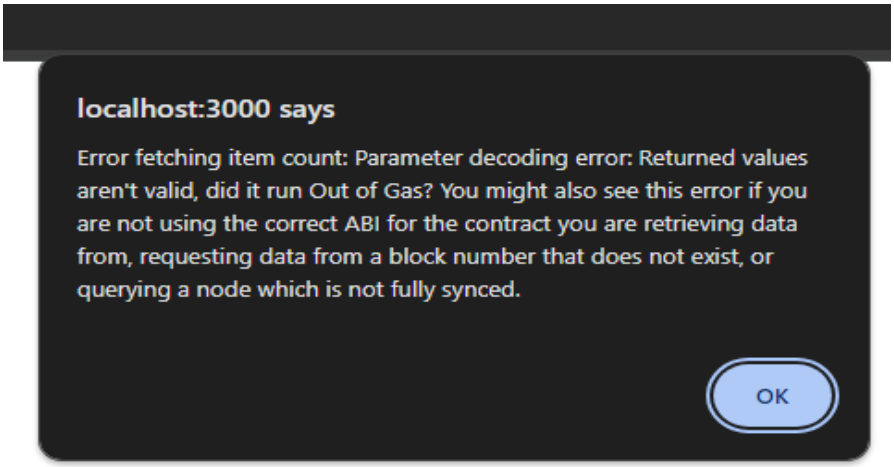
Slika 9. Logovi o deyplomentu ugovora na mrežu

Sljedeća stvar koju je potrebno napraviti jest otići na frontend dio aplikacije (i pokrenuti ju). Frontend s pametnim ugovorom komunicira putem web3.js datoteke koja izgleda ovako:

```
import Web3 from 'web3';
import simpleMarketplaceArtifact from '../../ignition/deployments/chain-
31337/artifacts/SimpleMarketplaceModule#SimpleMarketplace.json';

let web3;
let simpleMarketplaceContract;

// Postavljanje statičke adrese ugovora
const contractAddress = '0x5fbdb2315678afecb367f032d93f642f64180aa3';
const contractABI = simpleMarketplaceArtifact.abi;

if (typeof window !== 'undefined' && typeof window.ethereum !==
'undefined') {
  // Provjera okruženja preglednika i pokretanja MetaMask-a
  web3 = new Web3(window.ethereum);
  // Zahtjev za pristup MetaMask računu korisnika
  window.ethereum.request({ method: 'eth_requestAccounts' }).then(() => {
    simpleMarketplaceContract = new web3.eth.Contract(contractABI,
contractAddress);

    // Dohvaćanje broja stavki kao provjera
    simpleMarketplaceContract.methods.itemCount().call().catch(err => {
      alert('Error fetching item count: ' + err.message);
    });
  }).catch(err => {
```

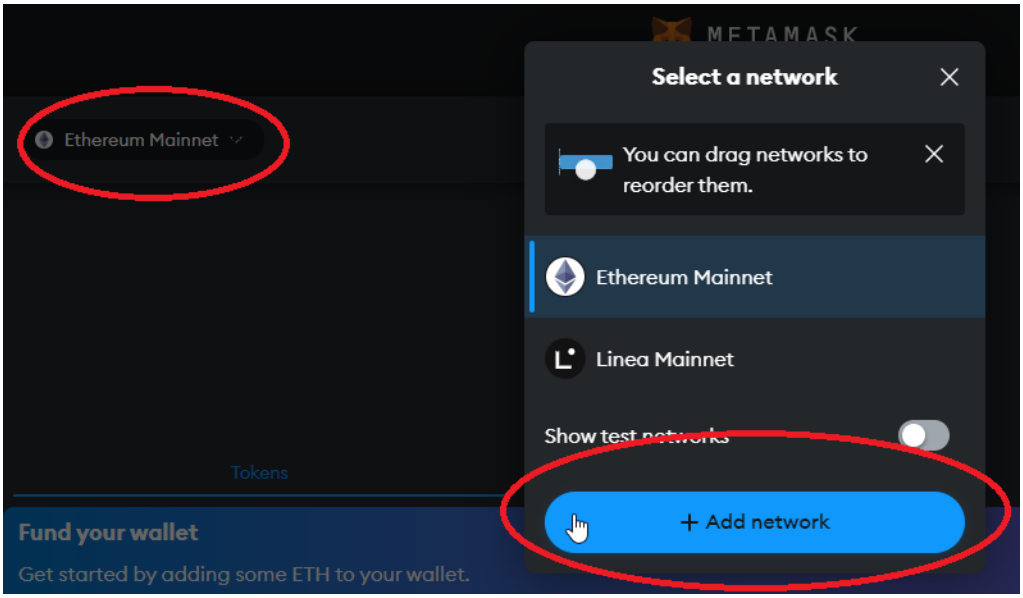
```
    alert('Error requesting MetaMask accounts: ' + err.message);
  });
} else {
  // Postavljanje na poslužitelju
  const provider = new
Web3.providers.HttpProvider('http://127.0.0.1:8545'); // Hardhat
  web3 = new Web3(provider);
  simpleMarketplaceContract = new web3.eth.Contract(contractABI,
contractAddress);

  // Dohvaćanje broja stavki kao provjera
  simpleMarketplaceContract.methods.itemCount().call().catch(err => {
    alert('Error fetching item count: ' + err.message);
  });
}

export { web3, simpleMarketplaceContract };
```

Bitno je unutar te datoteke postaviti ispravnu adresu (konstanta `contractAddress`) pametnog ugovora koji je deployan na mrežu, u suprotnom će aplikacija baciti sljedeću obavijest:



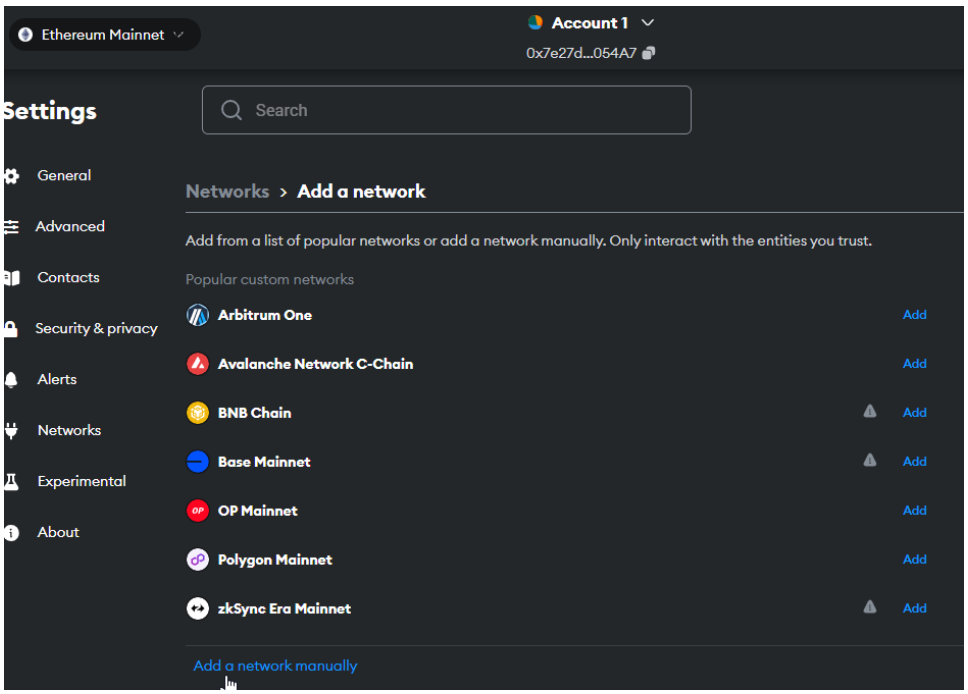
Slika 10. Poruka greške kada se ne postavi dobra adresa ugovora

Da bi se aplikacija mogla koristiti potrebno je skinuti MetaMask ekstenziju na preglednik koji se koristi. Nakon što se ekstenzija instalira potrebno je napraviti novi novčanik (ili koristiti postojeći) te manualno dodati Hardhat mrežu. Prvo se klikne gumb za mrežu na kojoj je korisnik trenutno (na slici je to u ovom slučaju Ethereum Mainnet) te potom se klikne gumb 'Add network' za ručno dodavanje mreže. Na slici ispod je označeno koje gumbove je potrebno kliknuti.



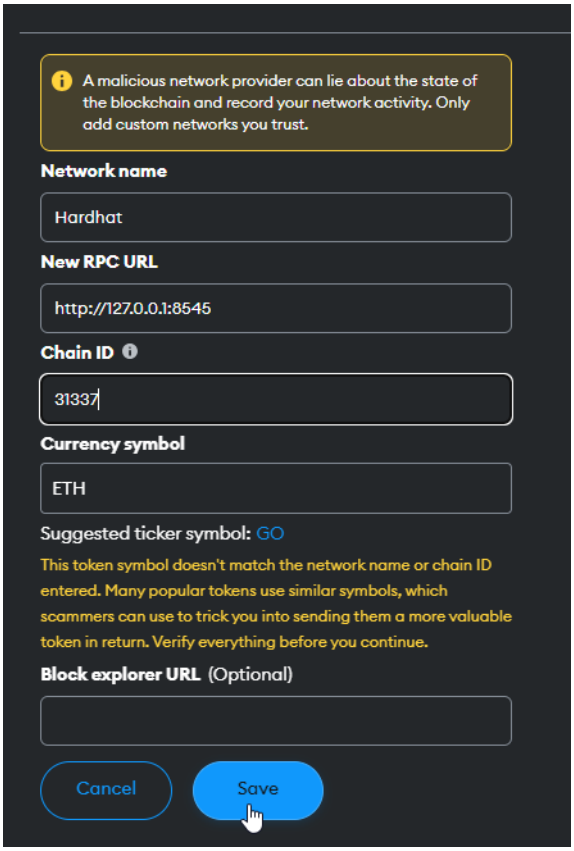
Slika 11. Gumbovi za dodavanje mreže

Nakon toga je potrebno kliknuti na gumb „Add a network manually“ koji se nalazi na dnu stranice.



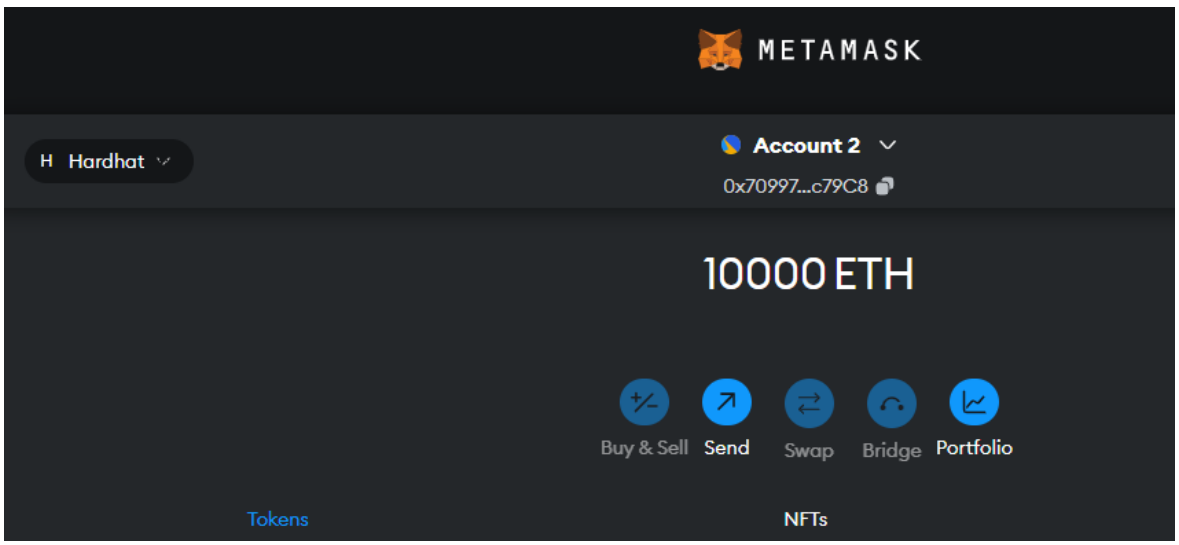
Slika 12. Gumb za dodavanje mreže ručno

Kod dodavanja mreže, naziv i simbol za valutu mogu biti proizvoljni, a ostatak je potrebno ispuniti kao na slici ispod.



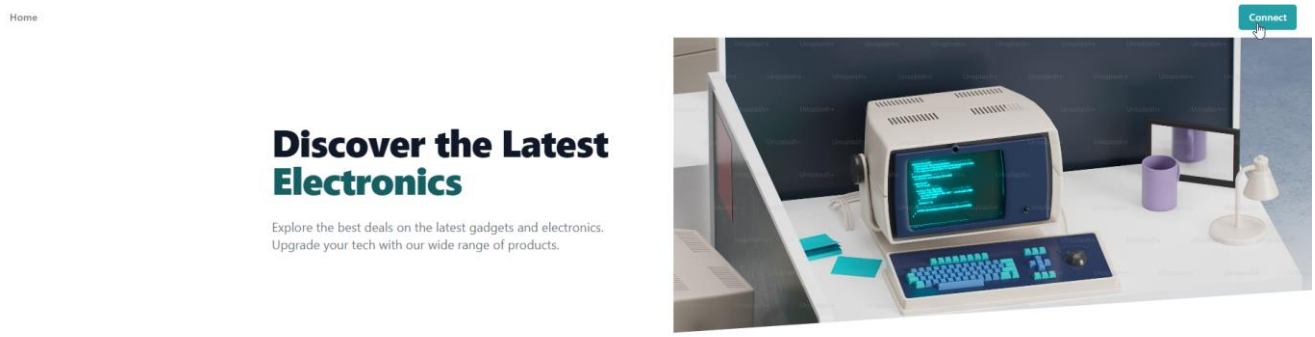
Slika 13. Podaci o mreži

Posljednja stvar koja se u MetaMasku treba napraviti jest importati adrese sa mreže. Kod biranja računa, treba se kliknuti 'Import account' te se iz terminala gdje se izvodi mreža treba kopirat privatni ključ za specifični račun. Kada se to napravi posljednja stvar koju je potrebno napraviti jest da se za taj račun izabere da koristi Hardhat mrežu koju smo postavili. Kad se to sve napravi, MetaMask bi trebao izgledati ovako:



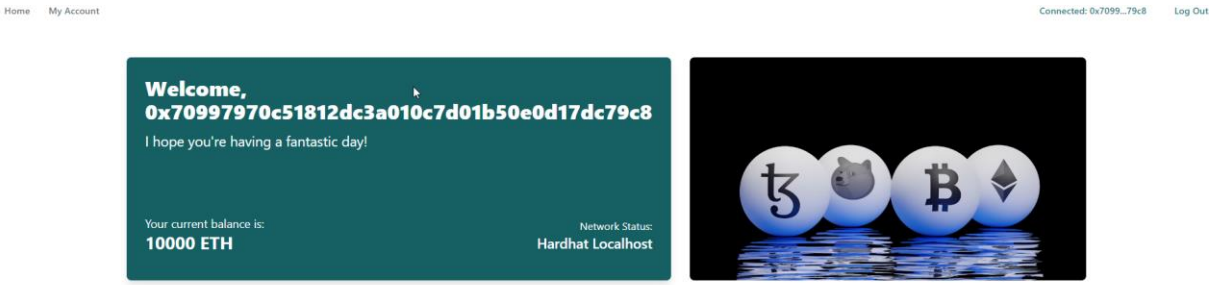
Slika 14. Uspješno dodani račun

Hardhat lokalna mreža je tako postavljena da svaki račun za testiranje ima početnih 1000 ETH. Treba se vratiti ponovno na localhost i kliknuti gumb „Connect“ kako bi se MetaMask povezao s lokalnom mrežom.



Slika 15. Prikaz početne stranice i gumba za povezivanje

Zatim se otvara posebno MetaMask i još nas jednom pita želimo li povezati localhost s njim, klikne se Next. Nakon što se MetaMask poveže, na navigaciji nam se pojavljuje „My Account“ koji kada se klikne pokazuje informacije o računu poput broja ETH, na kojoj mreži je povezan te adresu. Također na navigaciji se pojavljuje poruka: Connected: adresa novčanika te se nudi gumb za odjavu.



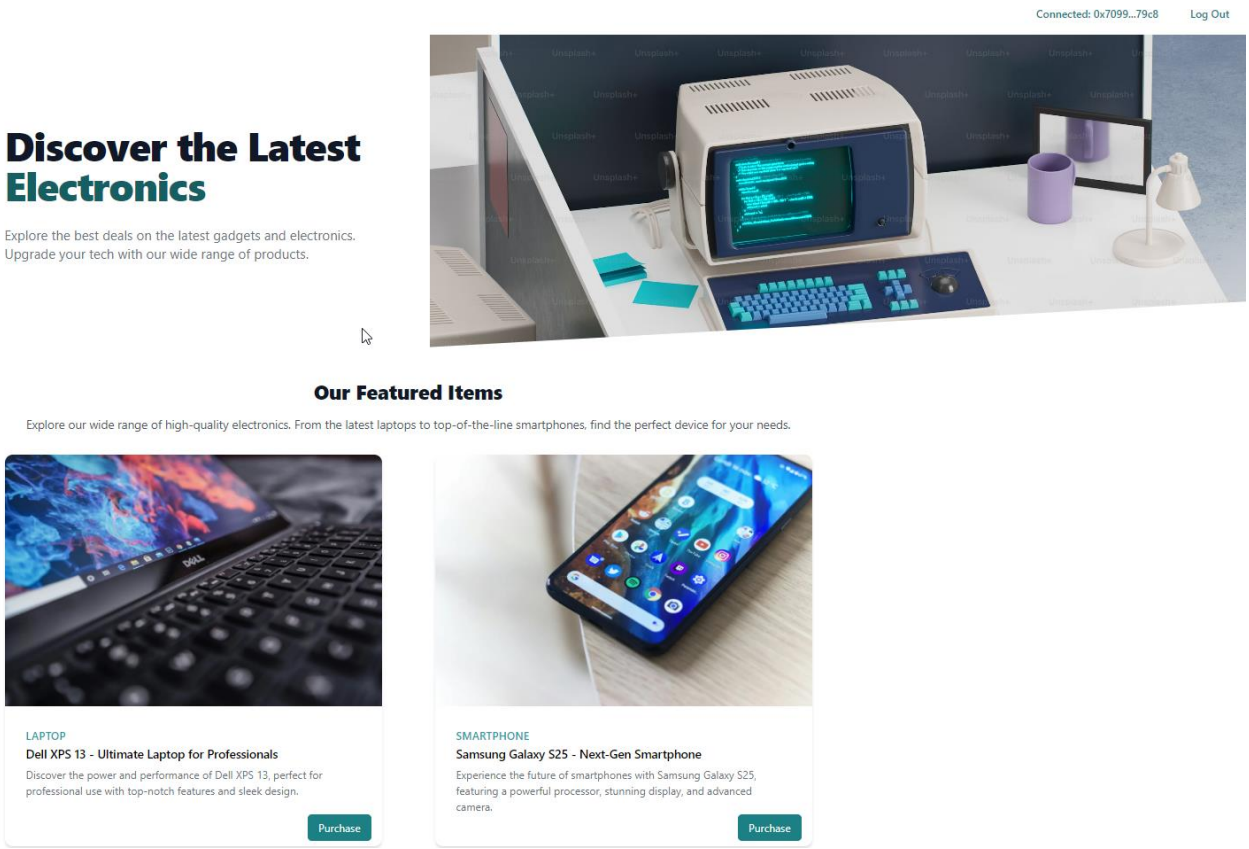
Slika 16. Stranica "My Account"

Logika koja stoji iza dohvaćanja informacijama o računu, nalazi se u useAccount.js te izgleda ovako:

```
import { useState, useEffect } from 'react';
import { web3 } from './web3';
import { useRouter } from 'next/router';

export const useAccount = () => {
    const [account, setAccount] = useState('');
    const [balance, setBalance] = useState('');
    const [network, setNetwork] = useState('');
    const [isLoggedIn, setIsLoggedIn] = useState(false); // Novo stanje za
status prijave
    const router = useRouter();

    // Funkcija za dohvaćanje imena mreže prema ID-u mreže
```

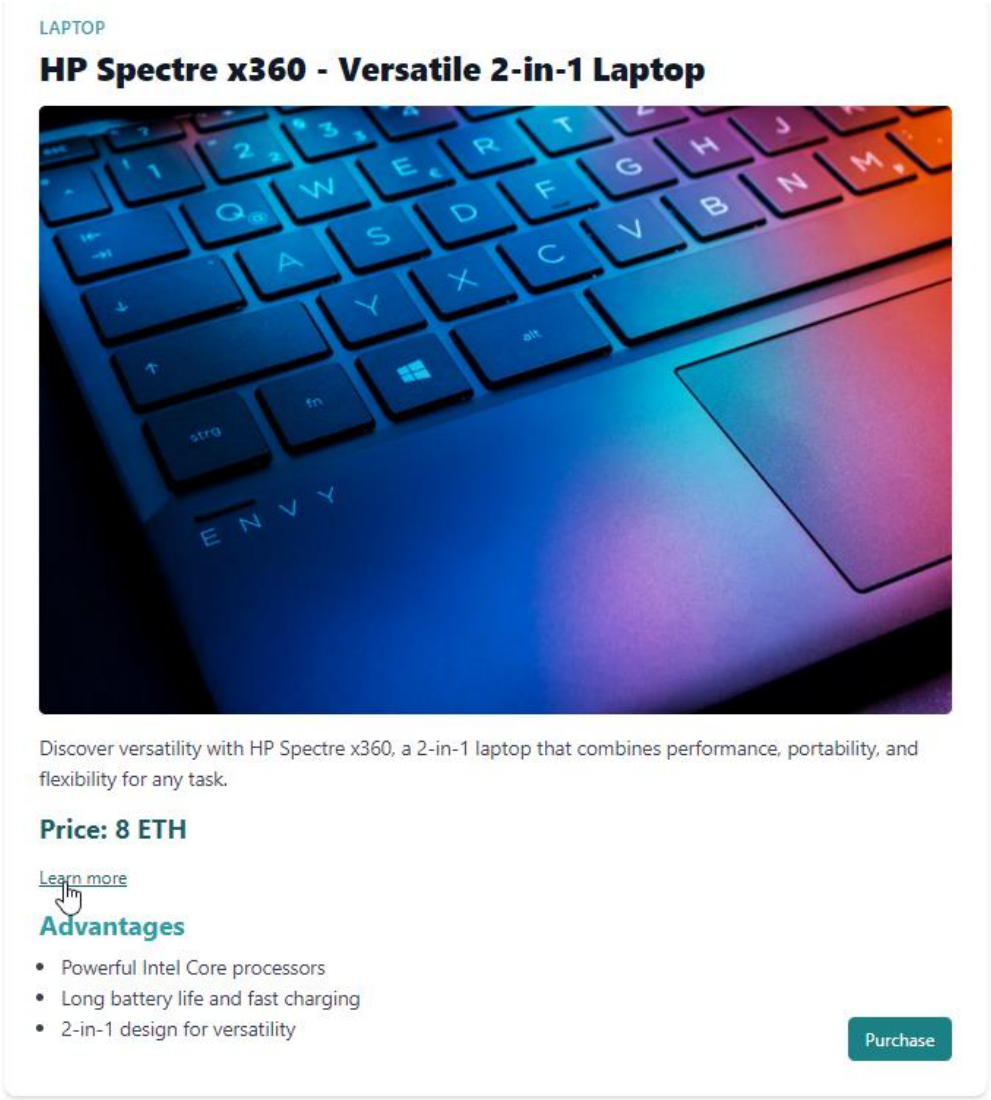
```
    const getNetworkName = (networkId) => {
        const networkNames = {
            1: 'Ethereum Mainnet',
            3: 'Ropsten Testnet',
            4: 'Rinkeby Testnet',
            5: 'Goerli Testnet',
            42: 'Kovan Testnet',
            31337: 'Hardhat Localhost',
            5777: 'Ganache',
            10: 'Optimistic Ethereum',
            11: 'Optimistic Kovan'
        };
        return networkNames[networkId] || `Unknown Network (${networkId})`;
    };

    // Funkcija za dohvaćanje detalja o računu
    const getAccountDetails = async () => {
        try {
            const accounts = await window.ethereum?.request({ method:
'eth_requestAccounts' }) ?? await web3.eth.getAccounts();
            if (accounts.length > 0) {
                const account = accounts[0];
                const balanceInWei = await web3.eth.getBalance(account);
                const networkId = await web3.eth.net.getId();

                setAccount(account);
                setBalance(web3.utils.fromWei(balanceInWei, 'ether'));
                setNetwork(getNetworkName(Number(networkId)));
                setIsLoggedIn(true); // Korisnik je prijavljen

            } else {
                setIsLoggedIn(false); // Korisnik nije prijavljen
                router.push('/');
            }
        } catch (error) {
            console.error("Error getting account details:", error);
            setNetwork("Error fetching network");
        }
    };

    // Korištenje useEffect za dohvaćanje detalja o računu prilikom
promjene router-a
    useEffect(() => {
        getAccountDetails();
    }, [router]);

    return { account, setAccount, balance, network, setNetwork, isLoggedIn,
getAccountDetails };
};
```

Početna web stranica izgleda kao na slici ispod. Za stiliziranje u Reactu koristila sam Tailwind CSS. Programski kôd za cijelu aplikaciju može se pristupiti na sljedećoj poveznici: https://github.com/sanja1999/diplomski



Slika 17. Početna stranica aplikacije

Da bi se saznale dodatne informacije o proizvodima koji su dostupni potrebno je kliknuti na naziv njihov te se potom otvara stranica za svaki proizvod koji dodatno navodi prednosti te cijenu proizvoda. Proizvod se može kupiti odmah s početne stranice ili sa stranice o specifičnom proizvodu.



Slika 18. Stranica o specifičnom proizvodu

Kao što sam već prije rekla, svi podaci koji se u ovoj testnoj aplikaciji prikazuju se čitaju iz specifične JSON datoteke (items.json). Kod za dohvaćanje podataka se nalazi u fetch.js datoteci te izgleda ovako:

```
import items from "./items.json";

export const getAllItems = () => {
    return {
        data: items,
        itemsMap: items.reduce((acc, item, index) => {
            acc[item.id] = item;
            acc[item.id].index = index;
            return acc;
        }, {})
    };
}
```

Za svaki artikl u nizu items, reduce funkcija poziva funkciju s tri parametra: acc (akumulator), item (trenutni artikl) i index (trenutni indeks u nizu). Unutar funkcije svaki artikl dodaje se u akumulator acc koristeći item.id kao ključ. Dodatno, indeks artikla u nizu items dodaje se kao svojstvo index unutar tog objekta.

Frontend je postavljen tako da početna stranica poziva dodatne komponente te da se unutar te stranice izvršava funkcija getAllItems() koja dohvaća informacije o proizvodima te koja je potrebna kako bi se poslije proslijedile te informacije na komponentu Catalog. Što se tiče kupovine, te kako se ona radi, potrebno je kliknuti gumb „Purchase“ koji potom pokreće funkciju handlePurchase gdje se odvija logika. Evo kako izgleda Catalog.js:

```
import Image from 'next/image';
import Link from 'next/link';
import { useAccount } from '../../../utils/useAccount';
import { handlePurchase } from '../../../utils/purchase';

export default function Catalog({ items }) {
    const { account, isLoggedIn } = useAccount(); // Korištenje hook-a za
dobivanje računa i statusa prijave

    return (
        <section className="max-w-7xl mx-auto py-12 px-4 sm:px-6 lg:px-8">
            <div className="mb-8 text-center">
                <h2 className="text-3xl font-extrabold text-gray-900">Our
Featured Items</h2>
                <p className="mt-4 text-lg text-gray-600">
                    Explore our wide range of high-quality electronics.
From the latest laptops to top-of-the-line smartphones, find the perfect
device for your needs.
                </p>
            </div>
            <div className="grid grid-cols-1 sm:grid-cols-2 gap-20">
                {items.map((item) => (
                    <div key={item.id} className="bg-white rounded-xl
shadow-md overflow-hidden flex flex-col relative">
                        <div className="h-50">
                            <div className="w-full h-full flex items-center
justify-center">
                                <div className="image-container flex items-
center justify-center">
                                    <Image
                                        src={item.coverImage}
                                        alt={item.title}
                                        width="0"
                                        height="0"
                                        sizes="100vw"
                                        className="w-auto h-auto"
                                    />
                                </div>
                            </div>
                        </div>

                        <div className="p-8 flex flex-col">
                            <div className="uppercase tracking-wide text-
custom-blue-300 font-semibold">{item.type}</div>
```

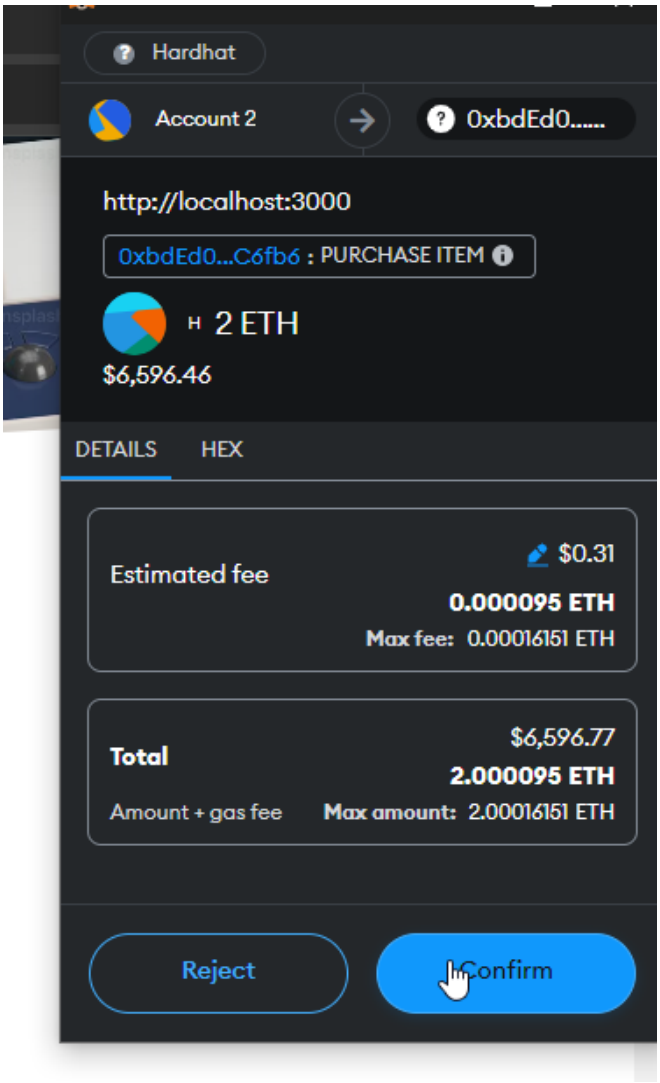
```
                            <Link legacyBehavior
href={`/items/${item.slug}`} passHref>
                                <a className="block mt-1 text-lg leading-
tight font-medium text-black hover:underline cursor-pointer">
                                    {item.title}
                                </a>
                            </Link>

                            <p className="mt-2 mb-4 text-gray-
500">{item.description}</p>

                            <div className="absolute bottom-2 right-2 mt-2
mr-2">
                                <button onClick={() =>
handlePurchase(item.id, item.price, account, isLoggedIn)}
                                    className="bg-custom-blue-400 text-white
px-4 py-2 rounded-md hover:bg-custom-blue-300">
                                    Purchase
                                </button>
                            </div>
                        </div>
                    </div>
                ))}
            </div>
        </section>
    );
}
```

Sva logika vezana uz handlePurchase funkciju nalazi se u purchase.js koji izgleda ovako:

```
import { web3, simpleMarketplaceContract } from './web3';

export const handlePurchase = async (itemId, price, account, isLoggedIn) =>
{
    if (!isLoggedIn) {
        alert('You are not logged in. Please log in to MetaMask.');
        return;
    }

    try {
        // Pretvorba cijene u Wei
        const priceInWei = web3.utils.toWei(price.toString(), 'ether');

        // Dohvaćanje stanja računa
        const balanceInWei = await web3.eth.getBalance(account);

        // Provjera ima li račun dovoljno sredstava
        if (parseInt(balanceInWei) < parseInt(priceInWei)) {
            alert('Insufficient balance to complete the purchase.');
            return;
        }

        await simpleMarketplaceContract.methods.purchaseItem(itemId).send({
            from: account,
            value: priceInWei
        });
    } catch (error) {
        console.error("Error purchasing item:", error);
```

```
    }
};
```

Funkcija prvo provjerava je li korisnik prijavljen, a ako nije, prikazuje poruku i zaustavlja izvršavanje. Zatim pretvara cijenu artikla iz ethera u wei te dohvaća stanje računa korisnika. Ako korisnik nema dovoljno sredstava za kupnju, prikazuje se poruka o nedovoljnom saldu i funkcija se zaustavlja. Ako su svi uvjeti zadovoljeni, funkcija poziva metodu purchaseItem iz pametnog ugovora te šalje transakciju s odgovarajućim iznosom wei iz korisnikovog računa. U slučaju greške, funkcija ispisuje poruku o pogrešci u konzolu. Kada se klikne gumb Purchase na frontendu, MetaMask otvara novi prozor te nas pita da potvrdimo transakciju.



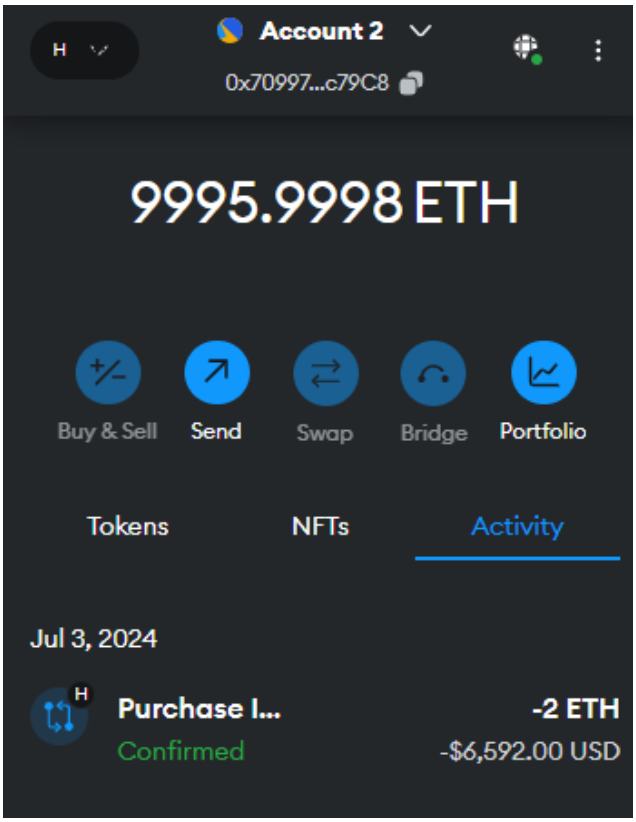
Slika 19. Poruka o tome želimo li potvrditi transakciju

Kada se klikne Confrim, MetaMask šalje obavijest o tome da je transakcija confirmed te potom izvršena. Na terminalu gdje je otvorena Hardhet lokalna mreža može se vidjeti obavijest da je transakcija izvršena.

```
eth_getBalance
net_version
eth_getBalance (21)
eth_gasPrice
eth_getTransactionCount
eth_sendRawTransaction
  Contract call:       <UnrecognizedContract>
  Transaction:         0xa74ab67872527c551539bf0911d409bc3195968e50cadae6e8ecdf606603bcca
  From:                0x70997970c51812dc3a010c7d01b50e0d17dc79c8
  To:                  0xbded0d2bf404bdcba897a74e6657f1f12e5c6fb6
  Value:               2. ETH
  Gas used:            37708 of 38002
  Block #3:            0x4ff4ecf8dd829c040ff68015292be8b1b74e83a7ef39454d47c67d859f118b7b
```

Slika 20. Log o izvršenoj transakciji

Također, kao potvrda da je transakcija uspješno izvršena, kada otvorimo MetaMask i informacije o računu, vide se podaci o transakciji te da se na računu smanjio iznos za koji je kupljen proizvod.



Slika 21. Informacije o računu i transakciji

7.3. Analiza isplativosti

Ova aplikacija predstavlja testni primjer kako pametni ugovori mogu komunicirati s lancem blokova mrežom i frontendom. Primarni cilj ove aplikacije je edukativne prirode, prikazujući osnovne principe razvoja decentraliziranih aplikacija (dApps) te interakcije s Ethereum lancem blokova putem MetaMask novčanika. Implementirani primjer, iako funkcionalan, služi samo kao dokaz koncepta.

Iako ovaj primjer uspješno demonstrira osnovnu funkcionalnost, u stvarnom svijetu aplikacija bi zahtijevala dodatna poboljšanja. Na primjer, korisničko sučelje bi trebalo biti intuitivnije i prilagođeno korisnicima bez tehničkog znanja. Potrebno je dodati više funkcionalnosti, napraviti više korisnika za aplikaciju, posebno administrator sustava. Sigurnosne mjere bi trebale biti rigoroznije kako bi se spriječile moguće zlouporabe uključujući dvofaktorsku autentifikaciju i dodatne provjere prilikom izvršavanja transakcija. Trenutno rješenje koristi Hardhat lokalnu mrežu za testiranje. U stvarnom svijetu, aplikacija bi trebala biti testirana i optimizirana za rad na glavnoj Ethereum mreži ili drugim skalabilnijim rješenjima poput Layer 2 rješenja (npr. Polygon, Optimistic Ethereum) kako bi se smanjili troškovi transakcija i poboljšala brzina. Također, s obzirom na brz razvoj tehnologije, važno je napomenuti da informacije o lancu blokova tehnologijama često brzo zastarijevaju. Primjerice, alat Ganache i Truffle sam počela koristiti jer je bio popularan u trenutku pretraživanja prijedloga teme rada, ali tehnologija i alati se stalno mijenjaju pa se ti alati više ne koriste. Ista stvar je s raznim modulima koji su dostupni za frontend koji komuniciraju s pametnim ugovorima, oni se izrazito brzo mijenjaju što može stvoriti izazov prilikom korištenja tehnologije.

8. Zaključak

Lanac blokova tehnologija, s posebnim naglaskom na Ethereum lanac blokova i pametne ugovore, predstavlja revolucionarni pristup digitalnim transakcijama i poslovnim procesima. Pametni ugovori omogućuju automatsko izvršavanje ugovornih obveza bez potrebe za posrednicima, čime se smanjuju troškovi, povećava efikasnost i transparentnost. Pravne izazove, uključujući Oracle problem, potrebno je riješiti kako bi se u potpunosti iskoristio potencijal pametnih ugovora.

Nadalje, razvoj decentraliziranih aplikacija (dApps) zahtijeva duboko razumijevanje tehnologija poput Solidity programskog jezika za pametne ugovore. Unatoč postojećim pravnim i tehničkim izazovima, potencijal lanac blokova tehnologije za transformaciju brojnih industrija je neosporan. Transparentnost, sigurnost i efikasnost koje donose pametni ugovori i decentralizirane aplikacije predstavljaju budućnost digitalne ekonomije. Daljnja istraživanja i razvoj u ovom području potiču inovacije i primjenu ove revolucionarne tehnologije u raznim industrijama, otvarajući nove mogućnosti za unapređenje poslovnih procesa i poboljšanje korisničkog iskustva. Posebno se ističe Web3, koncept koji se temelji na decentraliziranim tehnologijama lanca blokova. Web3 omogućuje stvaranje interneta nove generacije, gdje korisnici imaju veću kontrolu nad svojim podacima i interakcijama.

Zaključno, pametni ugovori i decentralizirane aplikacije imaju potencijal značajno transformirati različite industrije, nudeći sigurnija, efikasnija i transparentnija rješenja za poslovne procese. Njihova primjena može značajno unaprijediti financijske transakcije, upravljanje imovinom, zdravstvenu skrb i mnoge druge sektore. Daljnji razvoj i standardizacija ovih tehnologija, uz suradnju između tehnoloških inovatora i pravnih stručnjaka, bit će ključni za njihovu široku primjenu i integraciju u svakodnevni život.

Popis literature

[1] Simanta Shekhar Sarmah, “Understanding Blockchain Technology,” Computer Science and Engineering, vol. 8, br. 2, str. 23–29, 2018., pristupljeno: 16. lipnja 2024. [Online]. Dostupno na: http://article.sapub.org/10.5923.j.computer.20180802.02.html

[2] Anastasiia Lastovetska, “Blockchain Architecture Basics: Components, Structure, Benefits & Creation,” Mlsdev.com, 03. siječnja 2018. https://mlsdev.com/blog/156-how-to-build-your-own-blockchain-architecture (pristupljeno 16. lipnja 2024.).

[3] Huang, Qichen. (2023). Ethereum: Introduction, Expectation, and Implementation. Highlights in Science, Engineering and Technology. 41. 175-182. 10.54097/hset.v41i.6804.

[4] G. A. Oliva, A. E. Hassan, and Z. M. Jiang, "An exploratory study of smart contracts in the Ethereum blockchain platform," Empirical Software Engineering, vol. 25, pp. 1864–1904, 2020. doi: 10.1007/s10664-019-09796-5.

[5] „Ethereum roadmap | ethereum.org,” ethereum.org, 2024. https://ethereum.org/en/roadmap/ (pristupljeno 16. lipnja 2024.).

[6] Bitpanda, “Ethereum 2.0 simply explained,” Bitpanda.com, 2022. https://www.bitpanda.com/academy/en/lessons/ethereum-20-simply-explained/#phase-2-the-merge (pristupljeno 16. lipnja 2024.).

[7] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur i H. -N. Lee, "Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract," u IEEE Access, vol. 10, str. 6605-6621, 2022, doi: 10.1109/ACCESS.2021.3140091.

[8] De Filippi, P. & Wray, C. & Sileno, G., "Smart contracts," u Internet Policy Review, vol. 10, br. 2, 2021. https://doi.org/10.14763/2021.2.1549

[9] V. Y. Kemmoe, W. Stone, J. Kim, D. Kim i J. Son, "Recent Advances in Smart Contracts: A Technical Overview and State of the Art," u IEEE Access, vol. 8, str. 117782-117801, 2020, doi: 10.1109/ACCESS.2020.3005020.

[10] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin i F. -Y. Wang, "An Overview of Smart Contract: Architecture, Applications, and Future Trends," 2018 IEEE Intelligent Vehicles Symposium (IV), Changshu, Kina, 2018, str. 108-113, doi: 10.1109/IVS.2018.8500488.

[11] Taherdoost H., "Smart Contracts in Blockchain Technology: A Critical Review," u Information, vol. 14, br. 2, str. 117, 2023. https://doi.org/10.3390/info14020117.

[12] "Home," Oecd-ilibrary.org, 2018. https://www.oecd-ilibrary.org/sites/fbf2ebe9-en/index.html?itemId=/content/component/fbf2ebe9-en (pristupljeno 16. lipnja 2024.).

[13] “Solidity — Solidity 0.8.26 documentation,” Soliditylang.org, 2016. https://docs.soliditylang.org/en/v0.8.26/ (pristupljeno 16. lipnja 2024.).

[14] “React,” React.dev, 2015. https://react.dev/ (pristupljeno 16. lipnja 2024.).

Popis slika

Popis tablica

Prilozi