# Upravljanje stanjima unutar React biblioteke

**Anđić, Tomislav**

**Master's thesis / Diplomski rad**

**2021**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **University of Pula / Sveučilište Jurja Dobrile u Puli**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:137:347410

*Rights / Prava:* In copyright / Zaštićeno autorskim pravom.

*Download date / Datum preuzimanja:* **2024-04-20**

*Repository / Repozitorij:*

Digital Repository Juraj Dobrila University of Pula

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

**TOMISLAV ANĐIĆ**

# UPRAVLJANJE STANJIMA UNUTAR REACT BIBLIOTEKE

Diplomski rad

Pula, rujan, 2021 godine.

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

**TOMISLAV ANĐIĆ**

# UPRAVLJANJE STANJIMA UNUTAR REACT BIBLIOTEKE

Diplomski rad

**JMBAG: 0303038891, redoviti student**

**Studijskismjer: Informatika**

**Predmet: Izrada informatičkih projekata**

**Znanstveno područje: Društvene znanosti**
**Znanstveno polje: Informacijske i komunikcijske znanosti**
**Znanstvena grana: Informacijski sustavi i informatologija**

**Mentor: doc. dr. sc. Nikola Tanković**

Pula, rujan, 2021 godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani **Tomislav Anđić**, kandidat za **magistra informatike** ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, 01.09.2021 godine

IZJAVA
o korištenju autorskog djela

Ja, Tomislav Anđić dajem odobrenje Sveučilištu Jurja Dobrileu Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom **upravljanje stanjima unutar Reactbiblioteke** koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.
Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 01.09.2021

Potpis

_____

# UPRAVLJANJE STANJIMA UNUTAR REACT BIBLIOTEKE

**Tomislav Anđić**

**Sažetak:** U svrhu diplomskog rada obrađena je tema upravljanja stanja unutar React biblioteke gdje su se kroz teorijski i praktični rad prikazali najpopularniji načini upravljanja stanjima ove JavaScript biblioteke. React je JavaScript biblioteka za kreiranje klijentskih aplikacija, a u projektnom zadatku za upravljanje stanjima korištene su Redux, MobX i Recoil biblioteke te React Context API i lokalna stanja komponenti. U projektnom zadatku obrađen je i programski jezik JavaScript kao i sam pojam stanja i njegove vrste.

Sav izvorišni kod praktičnog dijela i sve implementacije po git granama dostupne su na sljedećoj GitHub poveznici: https://github.com/tomiandic/firmus.

Aplikacija je i objavljena preko Netlify servisa te joj se može pristupiti na sljedećoj poveznici: https://hopeful-nightingale-7509f4.netlify.app/

**Ključne riječi**: JavaScript, React, upravljanje stanjima, frontend, Redux, Recoil, MobX

# STATE MANAGEMENT IN REACT LIBRARY

**Tomislav Anđić**

**Abstract:** For the purpose of the thesis, the topic of state management within the React library was presented, where the most popular ways of state management of this JavaScript library were shown through the theoretical part and practical work. React is a JavaScript library for creating client applications, and the Redux, MobX, and Recoil libraries, as well as the React Context API and local component states, were used in the status management int the practical task. This thesis also deals with the JavaScript programming language as well as the very concept of the state and its types.

All source code from practical part and all implementations by git branches are available on the following GitHub link: https://github.com/tomiandic/firmus

Application code is also deployed over Netlify service and it can be accessed over the following link: https://hopeful-nightingale-7509f4.netlify.app/

**Keywords**: JavaScript, React, state management, frontend, Redux, Recoil, MobX

# Sadržaj


1. Uvod ..... 1
2.Javascript ..... 3
2.1 Uvod u Javascript ..... 3
2.2 Sintaksa i leksička struktura ..... 3
2.3 Paradigme programiranja u JavaScriptu ..... 4
2.3.1 Funkcionalno programiranje ..... 4
2.3.2 Objektno orijentirano programiranje ..... 5
2.4 Asinhronost ..... 6
2.5 Nasljeđivanje ..... 7
3.React biblioteka ..... 9
3.1 Uvod u react biblioteku ..... 9
3.2 Kompozicija ..... 10
3.3 React komponenta ..... 12
3.3.1 Klasna komponenta ..... 12
3.3.2 Funkcionalna komponenta ..... 13
3.3 Virtualni DOM ..... 14
3.4 JSX ..... 15
3.5 Tok podataka(stanja) u React-u ..... 16
4. Upravljanje stanjima ..... 17
4.1 Uvod ..... 17
4.2 Lokalna stanja ..... 17
4.3 Globalna stanja ..... 18
4.4 Flux ..... 19
4.5 Context API ..... 22
4.6 Redux ..... 23
4.6.1 Redux akcije ..... 25
4.6.2 Redux skladište ..... 25
4.6.3 Redux reduktori ..... 26
4.7 MobX ..... 26
4.7.1 Observable stanja ..... 27
4.7.2 Reakcije ..... 29
4.7.3 Akcije ..... 30
4.8 Recoil ..... 30

4.8.1 Atomi ..... 31
4.8.1 Selektori ..... 32
5. Projektni zadatak ..... 33
5.1 Uvod u izradu projektnog zadatka ..... 33
5.2 Proces izrade ..... 33
5.3 Aplikacija za registraciju studenata na servis za zapošljavanje „Firmus“ ..... 34
**6.Zaključak** ..... 41
Literatura ..... 42

# Popis slika

Slika 1. Usporedba izvođenja programa ..... 7
Slika 2. Prikaz kompozicije komponenti na UI-u ..... 11
Slika 3. Prikaz kompozicije komponenti stablom čvorova ..... 12
Slika 4. DOM reprezentacija ..... 14
Slika 5. Primjer JSX ugnježđivanja ..... 15
Slika 6. Primjer JSX korištenja „ClassName“ atributa ..... 16
Slika 7. Primjer JS izraza unutar JSX-a ..... 16
Slika 8. Jednosmjeni tok Flux arhitekture ..... 20
Slika 9. Prikaz iniciranja akcije (Bodouch) ..... 20
Slika 10. Prikaz uloge dispečera (Bodouch) ..... 21
Slika 11. Prikaz usporedbe toka stanja ..... 23
Slika 12. Prikaz omatanja aplikacije ReduxProvider komponentom ..... 24
Slika 13. Prikaz korištenja useSelector() hook funkcije u kodu ..... 24
Slika 14. Prikaz sintakse Redux akcije unutar koda ..... 25
Slika 15. Prikaz kreacije Redux skladišta unutar koda ..... 25
Slika 16. Prikaz sintakse reduktora unutar koda ..... 26
Slika 17. Prikaz kreacije observable stanja u MobX-u ..... 28
Slika 18. Prikaz međuodnosa observer i observable komponenti ..... 28
Slika 19. Prikaz toka događaja unutar MobX-a ..... 29
Slika 20. Prikaz kreacije atoma u kodu unutar Recoil-a ..... 31
Slika 21. Prikaz pretplate na atom u kodu ..... 31
Slika 22. Prikaz kreiranja selektora u kodu ..... 32
Slika 23. Prikaz pristupanja selektoru iz komponente u kodu ..... 32
Slika 24. Prikaz kompozicije aplikacije u obliku stabla ..... 34
Slika 25. Prikaz početne stranice aplilkacije ..... 35
Slika 26. Prikaz prvog koraka registracije ..... 36
Slika 27. Prikaz drugog koraka registracije ..... 37
Slika 28. Prikaz trećeg koraka registracije ..... 38
Slika 29. Prikaz četvrtog koraka registracije ..... 39
Slika 30. Prikaz zadnja tri koraka registracije ..... 40
Slika 31. Prikaz profila na zadnjem dijelu aplikacije ..... 40

# 1. Uvod

Eksponencialni razvoj internetskih platformi u prethodnom desetljeću uvelike je utjecao na razvoj novih riješenja problema koji se javljaju prilikom kreacije kompleksnih sustava na *webu* i upravljanjem velikih količina podataka unutar istih. Tema ovog diplomskog rada je upravljanje stanjima unutar React biblioteke, a ovaj rad bazirat će se na trenutno najpopularnijim i najkorištenijim riješenjima problema upravljanja stanja u klijentskim web aplikacijama. Osim obrade teme, rad se sastoji i od praktičnog dijela gdje se prikazuje način implementacije različitih načina upravljanja stanja u biblioteci React kroz jednostavnu klijentsku aplikaciju za registraciju studenata na servis za zapošljavanje.

JavaScript je unutar zadnjih deset godina postao jedan od najkorištenijih programskih jezika, a od svog nastanka prošao kroz mnoštvo iterativnih promjena te se od programskog jezika koji se koristi za upravljanje interaktivnih elemenata unutar web stranica razvio i transformirao u programski jezik sposoban za razvoj *full-stack* aplikacija. Ova popularnost, te sve veća kompleksnost aplikacija na web-u rezultiraju u brojnim JavaScript riješenjima u obliku programskih okvira i biblioteka koji postavljaju pravila i ograničenja sa ciljem ubrzanja razvoj aplikacija te izbjegavanja nedostataka koji se javljaju prilikom razvoja na *vanilla* način. Jedno od najpoznatijih takvih riješenja na klijentskoj strani aplikacije je React programska biblioteka, također poznat i kao ReactJS.

React je JavaScript biblioteka otvorenog kôda za izradu korisničkih sučelja tj. klijentskih aplikacija kreirana i održavana od strane Facebook-a i zajednice programera. Facebook je prva platforma koja je koristila React za implementaciju svojih korisničkih sučelja, a od kada je React postao biblioteka otvorenog kôda u 2013. godini, popularnost mu konstantno raste, te je danas korišten od strane mnoštva poznatih platformi kao što su Netflix, Twitter, Uber i dr.  Dio popularnosti React biblioteke leži u tome što omogućava da se složena korisnička sučelja razdijele na niz ponovno iskoristivih komponenti, a svaka od tih komponenti može imati svoje unutarnje stanje te reagirati na određene događaje. Stanja unutar

React aplikacije mogu se implementirati na više različitih načina te uz korištenje različitih biblioteka, a svaki od tih načina ima svoje prednosti i mane.

U drugom poglavlju ovog rada biti će obrađen programski jezik na kojem je React biblioteka bazirana, a to je JavaScript, nakon toga će u trećem poglavlju biti obrađena sama React biblioteka i svi njeni dijelovi i koncepti. U četvrtom poglavlju biti će obrađena tema upravljanja stanjima i najpoznatije vrste upravljanja stanjima unutar React biblioteke, a u zadnjem poglavlju biti će prikazan praktični rad te način njegove izrade i način korištenja.
Vrste upravljanja stanjima koje će biti obrađene u radu te koje su korištene u praktičnom radu su sljedeće:

- React lokalna stanja
- React Context
- Redux
- MobX
- Recoil

# 2.Javascript

JavaScript je programski jezik interpretacijskog tipa ponajprije namjenjen razvoju interaktivnih HTML-stranica. JavaScript kod se izravno izvršava bez provođenja u strojni jezik što znači da se interpretira, a ne kompajlira. On omogućuje implementaciju skripti koje će biti interaktivne s korisnikom iz vlastitog internet preglednika. Iako se često povezuju zbog sličnog imena, JavaScript nije pojednostavljena inačica programskog jezika Java niti ima ikakve veze sa ovim jezikom.

## 2.1 Uvod u Javascript

Javascript je jednodretveni *high-level* programski jezik kreiran 1995-te godine od strane američkog računalnog programera Brendan Eich-a. JavaScript se koristi na velikoj većini internetskih stranica, a danas ne postoji moderni internetski preglednik koji ne posjeduje mogućnost inerpretiranja JavaScript koda. Ovaj interpretni jezik baziran je na ECMAScript standardu, a omogućava više paradigmi od kojih su najčešće korištene funkcionalno i objektno orijentirano programiranje. ECMAScript standard, a samim time i JavaScript programski jezik definira minimalni API za rad sa primitivima kao što su npr. *string* i *number* te sa kompleksnijim strukturama podataka kao što su npr. liste, mape ili DOM-om(*DocumentObject Model*), ali ne uključuje nikakve *input/output* funkcionalnosti te ga njime opskrbljuju okruženja u kojima se izvodi, poput internet preglednika ili poslužitelja. U zadnjih deset godina otkako je izašao Node.js koji omogućava izvršavanje JavaScript koda izvan internetskih preglednika, raste popularnost izvršavanja Javascript koda u poslužiteljskom okruženju, sa programskim sučeljem koji JavaScript-u daje pristup operativnom sustavu te mogućnost manipulacije podataka, slanja i primanja podataka preko mreže te izvršavanja HTTP upita.

## 2.2 Sintaksa i leksička struktura

JavaScript je *case-sensitive* programski jezik što znači da se imena varijabli, funkcija i drugih identifikatora moraju pisati konzistentno sa pažnjom na velika i mala slova. Svaki identifikator u JavaScriptu mora započeti sa slovom, znakom doljnje crte (_) ili znakom dolara($), a prilikom definiranja identifikatora treba paziti jer u JavaScriptu

postoje određene rezervirane riječi kao što su *while* ili *for* koje se nesmiju koristiti u svrhu identifikatora. U JavaScriptu također nije obavezno pisati znak točka-zarez(;) na kraju svake izjave *(engl. Statement)*.(Flanagan, 2020)

U JavaScriptu postoje primitivni i objektni tipovi podataka, u primitivne tipove spadaju brojevi, tekst i *boolean* vrijednosti te specijalne JavaScript vrijednosti *null* i *undefined*. U JavaScriptu svaka vrijednost koja nije primitivni tip je objekt. Objektni tipovi su promijenjivi dok su primitivni nepromjenjivi *(engl. Immutable)* što znači da se objektni tipovi postavljaju po referenci dok se primitivni tipovi postavljaju po vrijednosti.

## 2.3 Paradigme programiranja u JavaScriptu

Paradigma programiranja je metoda pisanja koda ili način razmišljanja o izgradnji koda temeljenog na određenim načelima. Pisanje koda po određenoj paradigmi mora biti omogućena od strane programskog jezika za rješavanje određenog problema. Postoji više različitih paradigmi koje se koriste ovisno o problemu ili zahtjevima, a različite paradigme nisu međusobno isključive. Većina programskih jezika pruža mogućnost razvoja putem više paradigmi, a jedan od takvih jezika je i JavaScript. Prednost koju pružaju takvi jezici je programiranje gdje programer nije primoran odlučiti se striktno za jednu paradigmu, nego je u mogućnosti kombinirati više paradigmi unutar istog projekta.

Javascript podržava podskup imperativne i deklarativne paradigme programiranja, a to su sljedeće:

- Funkcionalna
- Objektno orijentirana
- *Event-driven*

### 2.3.1 Funkcionalno programiranje

Funkcionalna programiranje u novije vrijeme prati uzlaznu putanju korištenja sličnu objektno-orijentiranom programiranju prije par desetljeća te polako ulazi u sve moderne programske jezike. Isto tako se unutar React biblioteke implementira koncepte ove paradigme gdje ne samo da je moguće rezmišljati o komponentama u

smislu funkcija već ih je moguće i implementirati kao funkcije.

Funkcionalmo programiranje je podvrsta deklarativne paradigme programiranja, a ima svoje korijene u matematičkom formalnom sistemu računanja baziranom samo na funkcijama pod nazivom *lambda calculus*. Glavni principi ove paradigme su izvršavanje serija funkcija, gdje te funkcije nisu direktno povezane sa podacima i uvjek za iste ulazne podatke daju iste izlazne podatke ne proizvodeći nikakve programske nuspojave mijenjajući postojeće podatke. U funkcionalnom programiranju, logičke jedinice su smještene unutar funkcija, te funkcije je moguće koristiti za dobijanje željenog izlaza bez znanja o njihovom unutarnjem načinu kreiranja izlaza, sve što je programeru bitno je koji ulazi su neophodni za dobijanje željenog izlaza.Programer prilikom korištenja funkcionalnog programiranja prvo usmjerava razmišljanje na primjenu *high-level* apstrakcija, a nakon toga razmišlja o detaljima implementacije. Prednosti ovakvog razmišljanja na višoj razini apstrakcije su da potiču programera na kategorizaciju problema i uviđanje njihovih sličnosti, a osim toga programer se može koncentrirati na riješavanje relevantnih problema bez potrebe za definiranje *low-level* detalja(Ford, 2014).

U funkcionalnom programiranju, funkcije mogu biti vezane za neki identifikator, proslijeđene kao argument ili mogu biti povratna ekspresija*(engl. Return statement)* neke druge funkcije, s čime dijeli svojstva sa bilo kojim drugim tipom podataka. Korištenjem ovog načina programiranja se ne mijenja originalna struktura podataka, nego se kreiraju kopije originalne strukture sa željenim promjenama koje se onda koriste umjesto originala.

### 2.3.2 Objektno orijentirano programiranje

Objektno-orijentirano programiranje je ideja gdje objekti predstavljaju strukturu podataka koje je moguće manipulirati putem funkcija. Objekti su entiteti iz stvarnog svijeta koji se modeliraju unutar programa s čime se stvarni svijet preslikava unutar koda. Objekti mogu sadržavati podatke i funkcije što znači da su svi podaci i funkcije programaenkapsulirani unutar jednog ili više objekata.

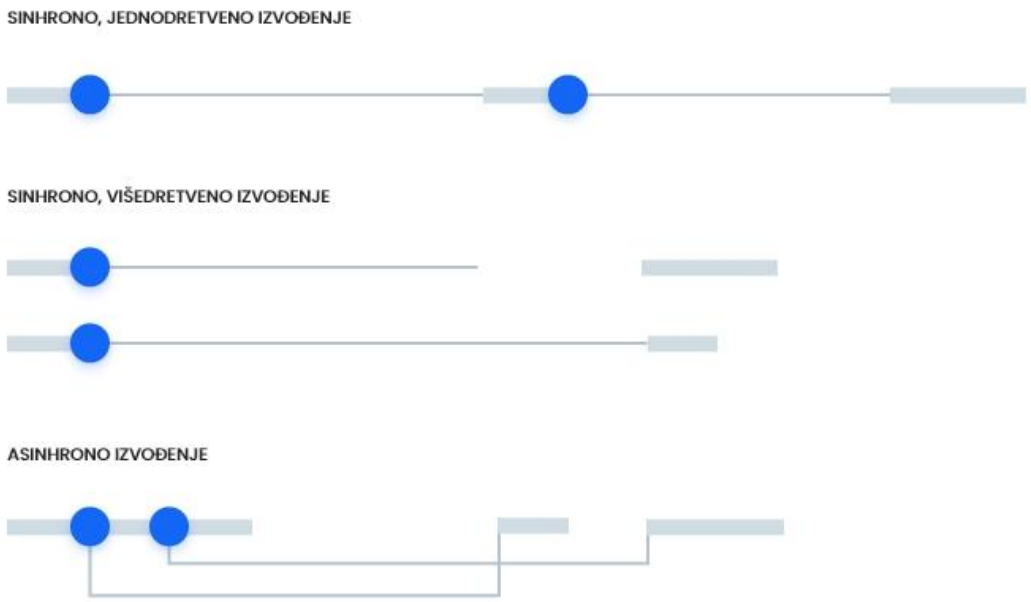
Za kreaciju objekta potrebna je klasa, klasa definira strukturu i atribute svakog objekta nastalog kao instanca te klase. Svaki objekt može imati jedinstvene podatke po strukturi definiranoj putem klase.

## 2.4 Asinhronost

U modernim klijentskim aplikacijama se sve češće javlja potreba za interakcijom sa resursima koji se nalaze izvan radne memorije. Dohvaćanje takvih resursa sa npr.poslužitelja ili tvrdog diska često zahtjeva relativno duži vremenski period u usporedbi sa čitanjem iz memorije.

U tradicionalnom sinhronom programskom modelu operacije se obavljaju sekvencijalno jedna za drugom te program prelazi na sljedeći korak nakon što se trenutni korak u potpunosti izvršio. Ovakvo linearno izvršavanje koda nailazi na problem prilikom dohvaćanja resursa za koji je potreban duži vremenski period gdje se program stavlja na čekanje. Jedno od riješenja ovakovog problema je višedretvenost*(engl. Multi threading)* gdje je moguće obavljati više paralelnih procesa u isto vrijeme, međutim moderni programski dizajn se sve više oslanja na korištenje asinhronog modela programiranja za omogućavanje izvršavanja više operacija u isto vrijeme. Asinhronost programa je vrsta izvođenja programa gdje se prilikom izvođenja, operacija stavlja na čekanje ukoliko joj je za izvršavanje potreban određeni resurs kojeg nema u datom trenutku, a dok taj resurs nije dostupan izvršavaju se druge operacije koje nisu nužno vezane za prvotnu operaciju.

Javascript je jednodretveni programski jezik, te nije u mogućnosti obavljati više paralelnih procesa u isto vrijeme, a zbog ovakvog načina izvršavanja,asinhronost je riješenje koje omogućava privid paralelnog izvršavanja procesa. Na slici 1. Prikazana je usporedba sinhronog i asinhronog izvođenja programa, na dijagramu sa slike deblja linija predstavlja normalno izvršavanje programa, a tanja linija predstavlja vrijeme čekanja na resurs. U sinhronom modelu, vrijeme čekanja na resurs je dio vremenske crte za pojedinu dretvu, dok se u asinhronom modelu sa početkom čekanja na resurs konceptualno odvaja vremenska crta gdje se program koji je inicirao akciju nastavlja izvršavati, a akcija dohvata resursa obavještava program kada se izvrši.




*Slika 1 Usporedba izvođenja programa*

Asinhrono programiranje je vrlo česta pojava u klijentskim aplikacijama gdje uporaba ovakvog načina izvršavanja uvelike doprinosi korisničkom iskustvu i toku gdje se korisnik ne susreće sa znakom učitavanja nakon svake obavljene akcije koja zahtjeva nekakav *input/output*.

## 2.5 Nasljeđivanje

Nasljeđivanje je važna komponenta u većini programskih jezika. Glavna prednost nasljeđivanja je ponovna uporaba istog koda, u objektno orijentiranom programiranju, ukoliko je jedna klasa slična drugoj, potrebno je samo specificirati razlike. Obrasci ponovne uporabe koda su iznimno značajni za smanjivanje vremena i troška razvoja aplikacije, a JavaScript podržava više načina nasljeđivanja. U programskim jezicima poput Java-e, objekti su instance klasa, a klasa se može naslijediti iz druge klase. JavaScript je prototipski programski jezik, što znači da se objekti mogu naslijeđivati iz drugih objekata. Svaki od kreiranih objekata sadrži „link“ na drugi objekt koji nazivamo njegovim prototipom, a taj prototipski objekt ima svoj prototip, i tako se ulančavaju dok se de dosegne objekt sa null vrijednosti kao prototip. Neke od vrsta prototipskog nasljeđivanja su:

- Diferencijalno nasljeđivanje – u difrencijalnom nasljeđivanju postoji *delegate* prototipski objekt koji služi kao baza za druge objekte. Kada se nasljedi *delegate* prototip, novi objekt dobije referencu na taj prototip. Kada se u novonastalom objektu pristupa varijabli objekta, prvo se provjerava u samom objektu, a nakon toga u svim naslijeđenim prototipskim objektima po redu naslijeđivanja dok se ne stigne do delegate objekta
  Ovakav način nasljeđivanja štedi memoriju jer je potrebna samo jedna kopija pojedine metode, a dostupna je među svim instancama koje je nasljeđuju. Glavna mana ovakvog pristupa je spremanje stanja, gdje ako se stanje spremi u obliku objekta ili liste i pokuša mijenjati pojedini dio tog objekta ili liste, on se mijenja u svakoj instanci koja je naslijedila prototip. Kako bi se ovakvo ponašanje izbjeglo, potrebno je napraviti kopiju stanja za svaku instancu objekta.
- Konkatenativno nasljeđivanje – proces kopiranja svojstava objekta sa jednog na drugi objekat bez zadržavanja međusobnih referenci
- Funkcionalno nasljeđivanje – kombinacija diferencijalnog i konkatenativnog nasljeđivanja gdje se funkcije koriste za obavljanje nasljeđivanja, a to omogućuje zatvaranje privatnih podataka unutar funkcionalnog opsega.

# 3.React biblioteka

React (također poznat kao React.js ili ReactJS) besplatna je JavaScript biblioteka otvorenog koda za izgradnju korisničkih sučelja ili komponenti korisničkog sučelja. Održavaju ga Facebook, kao i individualni programeri i korporativne zajednice. React se može koristiti kao osnova za razvoj mrežnih ili mobilnih aplikacija. Međutim, React funkcionalnosti same po sebi često nisu dostatne te obično zahtijeva korištenje dodatnih biblioteka za *routing* i druge funkcijonalnosti za kompleksne klijentske aplikacije.
Druga značajka je uporaba virtualnog objektnog modela dokumenta ili virtualnog DOM -a. React putem VDOM-a sprema strukturu elemenata u memoriju, izračunava razliku, a zatim učinkovito ažurira prikazani DOM u internet pregledniku. Taj se proces naziva koordinacija.
Prvi je put je korišten na Facebookov News Feed-u u 2011., a kasnije na Instagram aplikaciji 2012. godine. Postao je *open-source* na JSConf US u svibnju 2013. godine.

## 3.1 Uvod u react biblioteku

Grafička korisnička sučelja konstantno su u fazi evoluiranja, a važnost i vrijednost grafičkih sučelja se uvidjela kroz slučajeve kao što su popularnost iPhone uređaja čije je grafičko sučelje operacijskog sustava predstavilo revoluciju korisničkog iskustva na mobitelima. Korisničkim sučeljima se pridodaje sve više pažnje i konstantno se sakupljaju nova saznanja i provode nova istraživanja za postizanje optimalnog iskustva korisnika. Kako bi se na webu mogla realizirati ova saznanja te kreirati kompleksna sučelja na što jednostavniji način, u zadnjih10 godina pojavljuje se veliki broj biblioteka i programskih okvira koje nude svoja riješenja kroz razne načine implementacije. Jedna od najpoznatijih takvih riješenja za klijentske aplikacije današnjice je React biblioteka.

Osim promjena koje se tiču performansi i vizualnih promjena koje korisnik aplikacije može vidjeti direktno, drastično se promijenio i način izrade i strukturiranja web aplikacija. Dok se nekada za gotovo svaku akciju koja mijenja što preglednik prikazuje korisniku na web aplikaciji učitavala kompletno nova stranica i samim time tjerala korisnika na duže čekanje, danas se koristi aplikacijski model jedne stranice *(engl. Single Page Application)*. Ovaj model funkcionira na način da se web aplikacija

učitava samo jednom, a nakon toga se ovisno o akciji izmjenjuju različiti pogledi *(engl. View).* Osim prednosti u domeni korisničkog iskustva, ovakav model omogućava i jednostavniji razvoj aplikacije kroz mogućnost zadržavanje i manipuliranje globalnog stanja direktno u aplikaciji bez da se izgubi prilikom izvršavanja svake akcije.

React se koristi za kreiranje klijentske strane web aplikacije, a unutar MVC *(engl. Model-View-Contoller)*arhitekture predstavlja pogled. U svojim početcima korišten je samo od strane Facebook-a i Instagram-a, a od 2013. godine postaje biblioteka otvorenog koda nakon čega se počinje širiti unutar zajednice te mu popularnost raste i počinje se koristiti od strane mnoštvo popularnih digitalnih platformi. Osim toga u 2015 i 2016, izlazi veliki broj popularnih alata kao što su ReactRouter, Redux i Mobx s ciljem proširenja mogućnosti React-a u domenama gdje su funkcionalnosti koje dolaze uz samu biblioteke bile nedostatne, neefikasne ili nepostojeće.

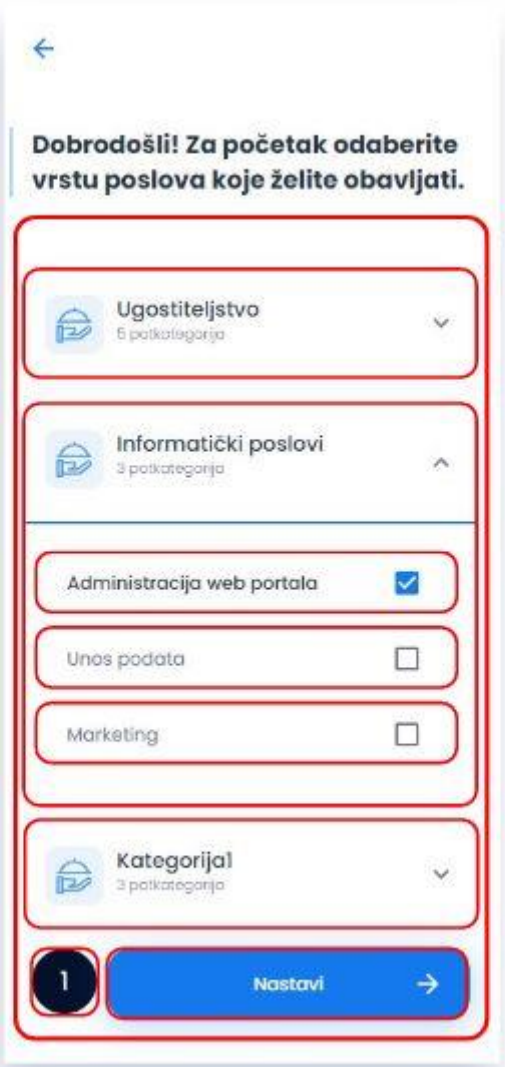
Popularnost ove biblioteke može se pripisati više faktora, ali jedne od glavnih su čisti i efektivni pristup programiranju. U Reactu, svaki dio grafičkog korisničkog sučelja je komponenta, svaka komponenta može biti kompozicija drugih komponenti, a sama aplikacija također predstavlja kompoziciju svih postojećih komponenti aplikacije. Iako programera ova arhitektura kompozicije komponenti može instinktivno navesti na razmišlja u smislu objektno-orijentiranog programiranja, React u biti koristi funkcionalnu tj. deklarativnu paradigmu, a u zadnje vrijeme se i sve više potiče korištenje funkcionalnih komponenti koje zamijenjuju dosad najčešće korištene klasne komponente.

## 3.2 Kompozicija

Jedna od glavnih značajka React-a je kompozicija komponenti. Komponente napisane na različit način od strane različitih programera i dalje mogu biti u istoj kompoziciji te međusobno funkcionirati. Kompozicija u Reactu je obrazac koji se može koristiti za „razbijanje“ velikih i kompleksnih komponenti u manje komponente i spajanje tih manjih komponenti u značajnu strukturu. Ovakav modularni pristup koristi se kako bi se postiglo što manje dupliciranog koda u sličnim komponentama te

omogućava ponovnu upotrebu istih koponenti bilo gdje unutar aplikacije.

Na slici 2. prikazana je kompozicija komponenti sučelja iz praktičnog dijela rada te je svaka zasebna React komponenta označena crvenim okvirom.



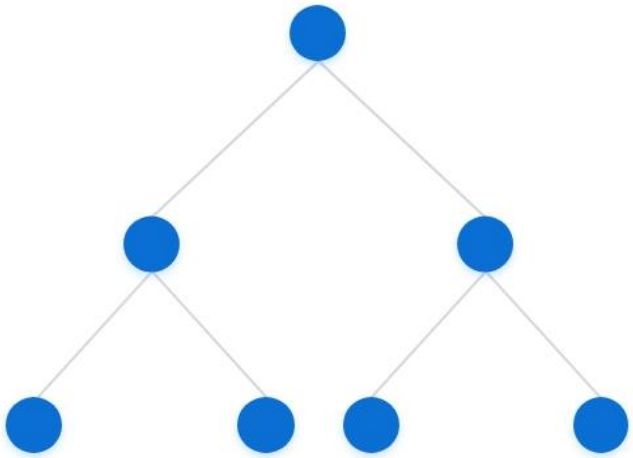

*Slika 2 Prikaz kompozicije komponenti na UI-u*

U React-u, kompoziciju možemo gledati kao hijerarhijsko stablo čvorova gdje svaka komponenta predstavlja jedan čvor, međuodnos komponenata unutar tog stabla definiraju sljedeća pravila:

- Nadređena komponenta *(engl. Parent component)* ne mora unaprijed znati svoje podređene komponente *(engl. Child component)*
- Podređene komponente nikada nemaju informaciju o svojim nadređenim

komponentama

- Podređene komponente nikada nemaju informaciju o drugim komponentama njihove nadređene komponente
- Veza među komponentama ima dobro definirano sučelje kroz tzv. props svojstva



*Slika 3 Prikaz kompozicije komponenti stablom čvorova*

## 3.3 React komponenta

Svako korisničko sučelje sastoji se od dijelova, a kada se ti dijelovi sastave u jednu kompoziciju stvara se korisničko sučelje. Koristeći aplikaciju, korisnik može pregledavati niz istih dijelove korisničkog sučelja koji mu prikazuju potpuno različite podatke. Svaku od tih dijelova u React biblioteci nazivamo komponentom. Komponente omogućavaju ponovnu uporabu iste već definirane strukture sa potpuno različitim podacima. Ovo svojstvo React-a čini ga vrlo skalabilnim riješenjem gdje se jedna komponenta može instancirati više puta na više različitih mijesta. React komponente se mogu promatrati kao funkcije, one omogućavaju kreiranje React aplikacije kroz kompoziciju, svaka od komponenti koja čini kompoziciju može biti neovisna i ponovno iskoristiva, te se o svakoj komponenti promišlja u izolaciji.

### 3.3.1 Klasna komponenta

Klasna komponenta je vrsta komponente koja se sintaksno definira kao JavaScript klasa, ima pristup stanjima i mogućnost modificiranja tih stanja pomoću *setState()*

metode.U ovakvoj vrsti komponenti također postoje predefinirane *lifecycle* metode koje se pozivaju ovisno o fazi izvođenja u kojem se komponenta trenutno nalazi. Lifecycle metode se koriste samo po potrebi komponente, a većina funkcionalnosti koje nude se bave provjerom ciklusa postavljanja komponente. Lifecycle metode koje postoje u klasnim komponentama su(Sidelnikov, 2016):

- *componentWillMount* – metoda pozvana prije *render()* metode koja se poziva samo jedan put tokom izvođenja komponente
- *componentDidMount* - metoda pozvana odmah nakon što se komponenta prikaže po prvi put
- *componentWillUpdate* - metoda koja se poziva svaki put prije nego što se izvrši *render()* metoda komponente
- *componentDidUpdate* - metoda koja se poziva svaki put nakon što *render()* metoda izvrši
- *componentWillUnmount* – metoda koja se poziva netom prije završetka izvršavanja komponente
- *shouldComponentUpdate* – metoda koja prima uvjet te ovisno o njemu ažurira komponentu

Iako su klasne komponente i dalje podržane u React-u one nisu preporučeni tip komponenti te se koriste sve manje, a kao zamijena se koriste funkcionalne komponente.

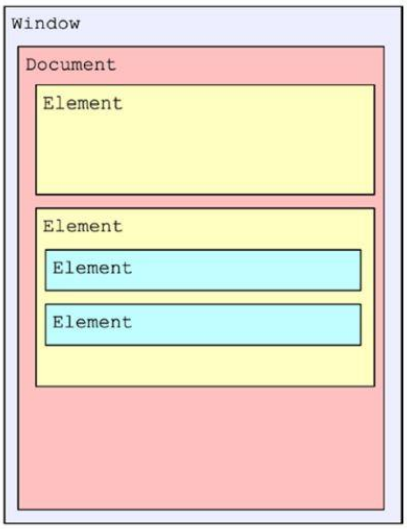
### 3.3.2 Funkcionalna komponenta

Funkcionalne komponente su ništa drugo nego JavaScript funkcije, mogu primati parametre, a povratna vrijednost im je JSX kod koji se transformira u DOM stablo. Funkcije su najjednostavniji način kreiranja komponenti u React-u. U usporedbi sa klasnim komponentama, ovoj vrsti komponenti fale određene značajke koje se nadomještavaju uporabom *hooks* funkcija kao što je *useState()* za inicializaciju i upravljanje stanja ili *useEffect()* koji nadomješta *lifecycle* funkcije. Osim toga props svojstva se u funkcionalnim komponentama definiraju kao argument funkcije, ali se koriste na isti način kao i u klasnim komponentama.

## 3.3 Virtualni DOM

Za razliku od tradicionalnih web aplikacija gdje se kod svakog izvršavanja neke akcije od strane klijenta na aplikaciji koja je uzrokovala veću promjenu UI-a, ponovno učitava cijelokupna stranica, ili se obavljao niz „skupih“ DOM modifikacija, u Reactu je ovaj proces puno efikasniji zahvaljujući virtualnom DOM-u.

Za shvatiti virtualni DOM prvo je potrebno razumijeti što DOM predstavlja. DOM *(engl. Document Object Model)* je hierarhijska reprezentacija grafičkog sučelja na web-u. U DOM-u, svaki HTML element predstavlja objekt, a svaki od tih objekata ima svoja svojstva*(engl. Properties)* i metode koji zajedno stvaraju DOM programsko sučelje. Tradicionalno je manipuliranje DOM-a putem JavaScripta bio spor proces i bilo je poželjno provoditi ga što rijeđe i samo kada je to neophodno. Kako bi se riješio ovaj problem neefikasne manipulacije DOM-a, React nudi svoje riješenje kroz VDOM.




*Slika 4 DOM reprezentacija*

Virtualni DOM omogućava nam brzi pristup relevantnim komponentama, te ažuriranje samo neophodnih DOM čvorova prilikom određene akcije. Ovaj način manipuliranja DOM-a omogućava brzinu korisničkog sučelja, web aplikacijama daje nativno korisničko iskustvo i smanjuje opterećenost mreže pri svakoj akciji.(Sidelnikov, 2016). Za razliku od DOM-a, virualni DOM je reprezentacija korisničkog sučelja koja

se nalazi u memoriji te je zbog toga vrlo brz za modifikaciju. React usporedbom stanja DOM-a i virtualnog DOM-a nalazi razlike te u skladu sa tim razlikama obavlja ažuriranje DOM-a kako bi korisničko sučelje održavalo trenutačna stanja.

## 3.4 JSX

Tradicionalni način kreiranja HTML elemenata putem JavaScripta zahtjeva veliki broj linija koda za kreiranje, dodavanje atributa, stila i pozicioniranje elementa u DOM stablu, ovaj način je također vrlo nečitljiv te neintuitivan. Kako bi riješili ovaj problem tvorci React biblioteke uvode ekstenziju pod nazivom JavaScript eXtension ili kraće JSX. JSX je React ekstenzija koja nam dopušta definiranje React elemenata koristeći sintaksu tag-ova koji oponašaju HTML tag-ove, a za razliku od HTML tag-ova mogu se koristiti direktno unutar JavaScript-a. JSX je još jedan način za kreaciju kompleksnih DOM stabala sa atributima, a olakšava razvoj i čitljivost koda zbog slične sintakse sa HTML-om. Internet preglednici nisu u stanju interpretirati JSX „direktno“ nego se JSX napisan od strane programera pretvara*(engl. Transpiles)* u standardni JavaScript putem kompajlera kao što je Babel prije nego što ga browser interpertira.(Sidelnikov, 2016)

Postoji niz razlika između HTML i JSX sintakse i niz limita kao što je nemogućnost korištenja potpuno svih atributa kao u HTML-u. JSX je u biti izvedenica XML jezika koji je striktniji od HTML-a. Neke od mogućnosti JSX-a koje nisu intuitivne programeru kod prelaska sa HTML-a na JSX su:

- Ugnježdene komponente – JSX omogućava ugnježđivanje podkomponenti*(engl. Childcomponents)* unutar nadkomponente*(engl. Parentcomponent)*kao što je prikazano na slici 5.

```
<IngredientsList>
  <Ingredient />
  <Ingredient />
  <Ingredient />
</IngredientsList>
```

*Slika 5 Primjer JSX ugnježđivanja*

- Korištenje „className“ atributa – u JavaScriptu je „class“ rezervirani izraz zbog čega je prilikom dodijeljivanja klase nekom elementu potrebno koristiti izraz „className“ kao što je prikazano na slici 6.

```
<h1 className="fancy">Baked Salmon</h1>
```

*Slika 6 Primjer JSX korištenja „ClassName“ atributa*

- JavaScript izrazi – JavaScript izrazi se u JSX-u pišu unutar vitičastih zagrada, unutar vitičastih zagrada se najčešće nalaze varijable čija se vrijednost prikazuje prilikom izvršavanja koda. Ukoliko se npr. želi prikazati *title* vrijednost unutar elementa, vrijednost se ubacuje putem JavaScript izraza unutar JSX-a na način prikazan na slici 7.

```
<h1>{title}</h1>
```

*Slika 7. Primjer JS izraza unutar JSX-a*

## 3.5 Tok podataka(stanja) u React-u

Postoje dva načina kako React komponenta može primiti podatke, a to su *top-down* i *sideways* način. Većina kompleksnih aplikacija koristi kombinaciju oba dva načina. *Top-down tj*. jednosmjerni *(engl. Unidirectional*) tok podataka znači da podaci kroz stablo komponenti uvjek teku od vrha stabla prema dnu stabla. To znači da svaka komponenta ima mogućnost kroz *props* svojstvo proslijediti svoje stanje samo svojoj podkomponenti. Sva stanja su uvjek posjed određene komponente, a bilo koji podaci izvedeni iz tih stanja mogu utjecati samo na podkomponentu ispod te komponente u stablu. Korištenjem *top-down* toka komponente imaju sljedeće osobine:

- Jednostavne su za modeliranje kao *pure* komponente
- Prenosive i ponovno iskoristive u aplikaciji
- Jednostavne za testiranje

Osim *top-down* toka postoji i *sideways* tok podataka, ovo je koncept gdje se podaci dostavljaju direktno u određenu komponentu, a podaci nisu dosli od nadkomponente. Ovakav tok omogućava bolju optimizaciju gdje se ponovno ažuriranje *(engl. Re-rendering)* komponenti ne obavlja od korijenske komponente već se samo komponenta kojoj se dostavljaju podaci ponovno ažurira. Korištenjem *sideways* toka komponente imaju slijedeće osobine:

- Upravljaju ciklusima ponovnog ažuriranja svojih podkomponenti
- Izolirani su od ponovnog ažuriranja nadkomponent

# 4. Upravljanje stanjima

Koncept stanja nije vezan za određeni programski jezik ili biblioteku, a stanja aplikacije se definiraju kao svi podaci unutar aplikacije koji postoje u memoriji u trenutku izvršavanja aplikacije, a u to mogu spadati sve varijable, objekti, fontovi, animacije, DOM čvorovi i sl.

## 4.1 Uvod

U React-u stanja definiramo kao niz vrijednosti koje kontroliraju ponašanje komponenti kroz vrijeme. Stanja se upravljaju unutar komponente kao i bilo koja druga varijabla, međutim za razliku od običnih varijabli koje „nestaju“ nakon izvršenja njihovih funkcija, stanja se „održavaju“*(engl. Persist)* dokle god se aplikacija izvodi. Postoji više načina upravljanja stanjima unutar React-a, od držanja stanja lokalno unutar komponente do korištenja raznih ekstenzija koje služe za efikasno upravljanje globalnim stanjima.
Kada se spominje upravljanje stanjima pri tome se misli se na inicijalizaciju, modificiranje i brisanje stanja.
Još jedna od važnih stvari koju treba napomenuti kod stanja u React-u je njihova nepromjenjivost, a ono što se misli pod time je da kada npr. objekt koji predstavlja stanje u Reactu potrebno modificirani on se ne mijenja „direktno“ već se kreira novi objekt na osnovu postojećeg koji se modificira te se on postavlja umjesto stare instance. Ovo svojstvo nepromjenjivosti stanja omogućava lakše predviđanje događaja prilikom modificiranja stanja.

## 4.2 Lokalna stanja

U React-u lokalno stanje omogućava instanciranje JavaScript objekta za držanje vrijednosti komponente koje mogu utjecati na njeno prikazivanje *(engl. Rendering*). Lokalna stanja se drže u izolaciji unutar komponente i nisu dostupna u drugim komponentama za čitanje i mijenjanje osim ako ih eksplicitno ne proslijedimo uporabom „props“ sučelja. U klasnim komponentama kada god se lokano stanje promijeni kroz korištenje *setState()* metode, React okida ažuriranje te komponente i svih nadkomponenti u hijerarhijskom stablu.

Korištenje lokalnog stanja dostatno je za manje komponente i/ili aplikacije ali uzmemo li u obzir česti slučaj kod malo kompleksnijih aplikacija gdje više komponenti na različitim mjestima u React hijererhijskom stablu moraju dijeliti ista stanja tu nailazimo na nedostatak lokalnih stanja. U nekim slučajevima ovaj problem moguće je riješiti preseljenjem stanja u nadkomponentu koja je zajednička podkomponentama koje moraju dijeliti ista stanja, ovakvo riješenje ubrzo postaje nedostatno i nepraktično kada su komponente koje moraju dijeliti stanja na različitim razinama u hijerarhijskom stablu te je potrebno slati „props“ parametre kroz više razina.

## 4.3 Globalna stanja

Ako se za demonstraciju globalnih stanja uzme jednostavni primjer internet trgovine u kojoj klijent želi filtrirati listu proizvoda te se obavljanjem akcije na filter komponenti mijenjaju rezultati u komponenti za listu proizvoda. U ovom slučaju oba dvije komponente moraju dijeliti ista stanja kako bi filter komponenta bila u mogućnosti mijenjati ta stanja, a komponenta liste proizvoda mogla znati kada filtrirati listu s obzirom na promjenu stanja od strane korisnika. Kada bi se komponenti koja lista proizvode dodalo i mogućnost stavljanja proizvoda u košaricu direktno iz liste, gdje onda i komponenta košarice također mora dijeliti ova ista stanja, stvari ubrzo postaju kompleksne po pitanju upravljanja stanja ukoliko se razmišlja sa premisom korištenja lokalnih stanja. Zbog ovakvih problema na koje se često nailazi prilikom razvoja kompleksnijih aplikacija uvode se globalna stanja. Globalna stanja su kako im i samo ime otkriva dostupna globalno unutar komponenti aplikacije. U teoriji, sve komponente unutar aplikacije imaju pristup globalnim stanjima ali u praksi to često nije poželjno zbog čega je na programeru da specificira koje komponente imaju pristup kojim stanjima.

Postoji niz biblioteka i načina kako dodati globalna stanja u React aplikaciju, najčešće se odluka donosi sa obzirom na vrstu aplikacije koja se razvija. Korištenjem globalnih stanja se uglavnom implementira sideways tok podataka.

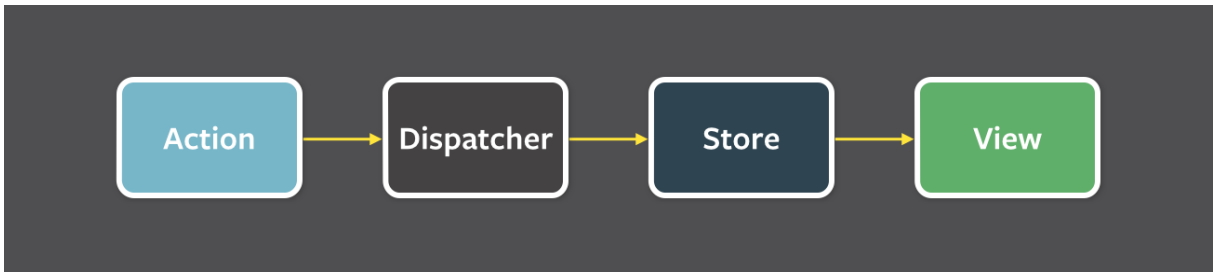
## 4.4 Flux

Flux je arhitekturni obrazac za mrežne aplikacije kreiran od strane Facebook-ovog tima koji je nakon svog izlaska uvelike utjecao na način upravljanja stanjima u klijentskim aplikacijama. Osim same Facebookove implementacije, izašlao je niz drugih biblioteka za upravljanje stanjima koje su implementirale ovu arhitekturu na svoj način. I dok je u samim početcima postojalo zasićenje ovih biblioteka, nije bilo potrebno puno vremena da se Redux iskristalizira kao najpopularniji odabir.

Flux je nastao kao alternativa tada prevladavajućoj MVC *(Model View Controller)* arhitekturi populariziranoj od strane raznih programskih okvira kao što su Backbone, Angular i Ember. Iako svaki od nabrojanih i mnogih drugih okvira koje koriste MVC obrazac imaju različite implementacije, većina dijele sličnu bolnu točku, a to je da tok podataka između modela, pogleda i kontrolera brzo postaje težak za praćenje.

U MVC arhitekturnom obrascu najčešće se koristi dvosmjerno vezivanje podataka gdje promjene u pogledu ažuriraju odgovarajuće modele, a isto tako promjene u modelima ažuriraju poglede. Ovakvo okidanje događaja*(engl. Events)* gdje pogled može ažurirati modele, a te promjene u modelima onda mogu ažurirati više drugih pogleda često stvaraju nedovoljno čistu sliku o implicitnim nuspojavama prilikom izvršavanja određene promjene.

Jednosmjerno vezivanje podataka Flux arhitekture rješava nabrojane probleme nepredvidljivosti stanja i problema koji nastaju prilikom usko vezanih modela i pogleda. Kod Flux-a, sve promjene na stanjima uzrokuju jednosmerni tok događaja. Uzme li se jednostavni primjer korisnika aplikacije koji preko sučelja stisne gumb koji obavlja neku promjenu u stanjima preko grafičkog sučelja, nakon stiska gumba se prvo akcija šalje na aplikacijski dispečer, dispečer šalje podatke u skladište podataka *(engl. Data Store)* gdje se oni ažuriraju, a nakon što se spremište ažurira, pogledi postaju svijesni novih podataka te ih prikazuju.(Garreau i Erikson, 2018). Ovaj jednosmjerni tok događaja prikazan je na slici 8.

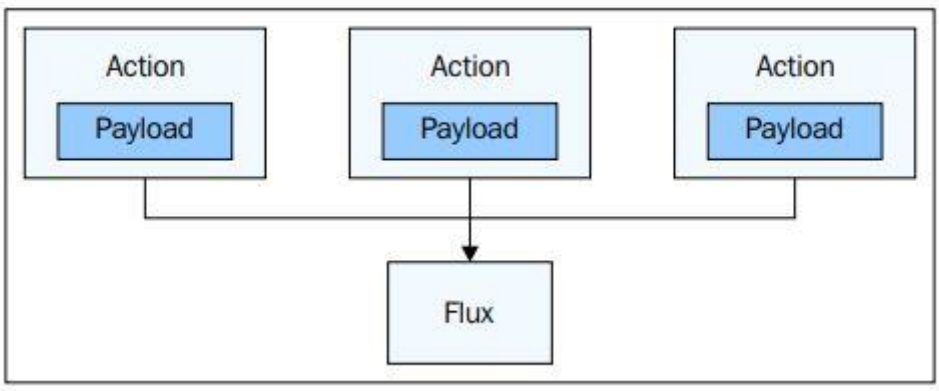



*Slika 8 Jednosmjeni tok Flux arhitekture*

Svi podaci prolaze kroz dispečer koji predstavlja središnje čvorište, dispečer se opskrbljuje akcijama, a najčešći izvor ovih akcija u klijentskim aplikacijama su korisnikove interakcije sa grafičkim sučeljem. Nakon toga dispečer poziva callback funkciju na koju se spremišta podataka prethodno registriraju, slanjem akcija svim registriranim spremištima. Unutar svojih registriranih callback funkcija, spremišta odgovaraju bilo kojoj akciji koja je relevantna stanjima koje održavaju. Spremišta nakon toga emitiraju *change* događaj kako bi obavijestili poglede da se desila promjena na podatkovnoj razini. Pogledi „slušaju“ ovakve događaje te dohvaćaju podatke iz spremišta, kada se desi promjena na pojedinom pogledu, obavlja se ažuriranje tog pogleda.

### 4.4.1 Akcije

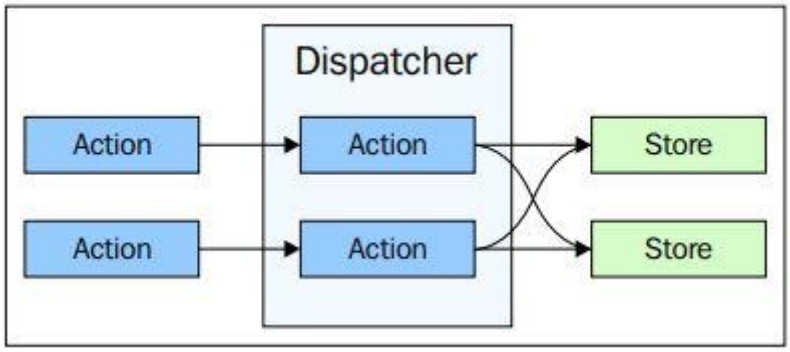
U Flux arhitekturi svaka promjena stanja započinje sa nekom akcijom. Akcija je uglavnom definirana kao obični objekt koji se sastoji od imena za identifikaciju akcije i podataka koje se želi proslijediti, a konceptualno ona predstavlja neki događaj unutar aplikacije kao npr. „Flitriraj listu korisnika“. Akcije se generiraju od strane korisnikove interakcije ili od strane interakcije sa poslužiteljom.(Boduch, 2016)




*Slika 9 Prikaz iniciranja akcije (Bodouch)*

### 4.4.2 Dispečer

Dispečer sam po sebi ima malo funkcionalnosti, a njegova glavna zadaća je primanje akcija i slanje tih akcija u spremišta podataka koja su se pretplatila na određenu akciju. Dispečeri su mjesto gdje se callback funkcije spremišta podataka registriraju i gdje se riješavaju ovisnosti o podacima. Spremišta obaviještavaju dispečer o drugim spremištima o kojima ovise te je na dispečeru da osigura da su podaci o kojima ovise dostupni.(Bodouch, 2016)




*Slika 10 Prikaz uloge dispečera (Bodouch)*

### 4.4.3 Skladišta

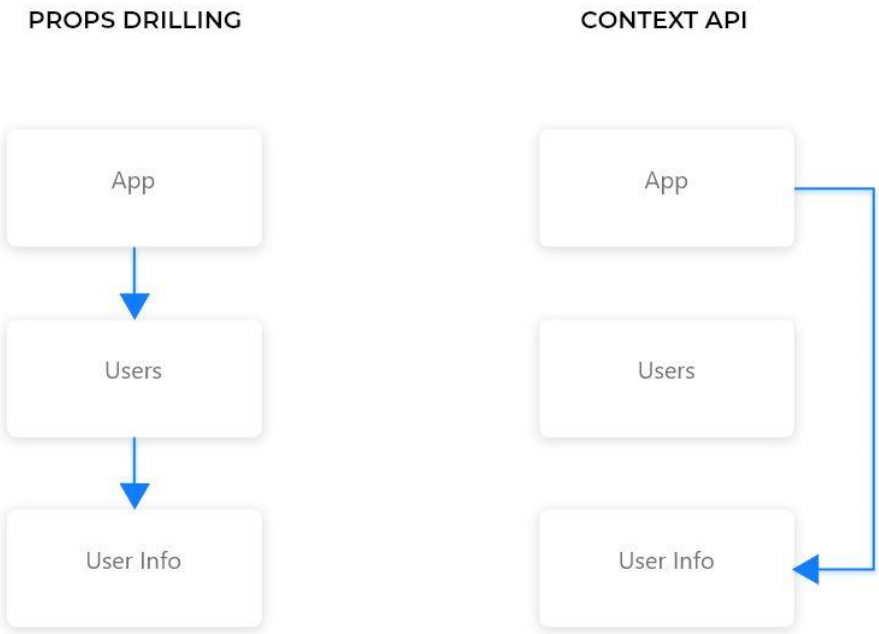
Skladišta su mjesto gdje se se drže stanja u Flux arhitekturi. Glavni aspekt promjene stanja u skladištima je da ne postoji nikakva vanjska logika koja determinira kada se mijenjanje mora izvršiti već je sve definirano u skladištima. Svako skladište u pravilu upravlja stanjima jedne domene u aplikaciji pa ako se za primjer uzme kreiranje skladišta internet trgovine u pravilu se kreira posebno skladište za košaricu i skladište za npr. proizvode.

## 4.5 Context API

Prilikom razvoja klijentske aplikacije u React-u, gledajući na aplikaciju iz perspektive stabla onda je vrlo čest slučaj da što je veća aplikacija to je korijen stabla udaljeniji od krajnjih elemenata te su dijeljena stanja koja se drže u korijenu sve dalje od komponenti kojima su potrebne za rad. Koristeći lokalna stanja, u tim slučajevima potrebno ih je provesti kroz više razina hijerarhijskog stabla da bi krajnje komponente dobile pristup potrebnim podacima. Ovakav pristup nije skalabilan i tjera programere na pisanje ponavljajućeg koda za propagiranje stanja kroz sve komponente koje služe kao „most“, a osim toga stvara se nepotrebna veza među ulančanim komponentama.

Zbog ovog problema sa stanjima React tim od verzije 16.3 uvodi Context API kao standardni dodatak unutar React biblioteke. Context API u React-u služi za omogućavanje dijeljenja stanja kroz sve komponente unutar aplikacije bez potrebe za *props* sučeljem. Context API se preporuča koristiti samo za stanja koja se koriste globalno, a ne mijenjaju često, vrlo je učinkovit za korištenje u svrhe poput propagacije aplikacijske teme, jezika ili autentifikacije(Facebook Open Source, n.d).

Context se koristi na način da se sva stanja postavljaju u *context provider* komponentu koja služi kao omotač *(engl. Wrapper)* za cijelo hijerarhijsko stablo aplikacije, a svaka komponenta koja ima pristup tim definiranim stanjima omotana je *context consumer* omotačem. Kada se context provider stanja mijenjaju, sve komponente koje su omotane context consumer komponentom se ažuriraju, ponašanje je slično kao i kada se mijenjaju props svojstva komponente. Problem optimiziranog izvođenja aplikacije sa Context API-jem se javlja kada imamo niz komponenti koje koriste jedan dio *context provider* stanja, i svakom promjenom samo jednog dijela stanja sve komponente omotane *context consumer* omotačem se ponovno ažuriraju. Zbog ovakvog načina rada Context-a, preporuča ga se koristiti samo kada je vrsta stanja koja se drži u provideru takva da se rijetko mijenja i kad se mijenja da ima utjecaj na sve komponente.




*Slika 11 Prikaz usporedbe toka stanja*

## 4.6 Redux

Redux je JavaScript biblioteka otvorenog koda kreirana od strane Dan Abramov-a i Andrew Clark-a, a služi za upravljanje stanjima aplikacije. Najčešće se koristi uz biblioteke kao što je React iako je neovisna o biblioteci ili okviru sa kojim se integrira. Redux je biblioteka inspirirana Flux arhitekturom te se sastoji od svih prednosti koje pruža ova arhitektura, a u isto vrijeme je oslobođen niza nedostataka Flux-a. Redux također koristi jednosmjerni tok podatak gdje promjene stanja uvjek kreću od akcija koje se otpremaju i ovisno o tipu akcije reduktor funkcije mijenjaju stanje.
Glavne razlike Redux arhitekture naspram Flux-a su:

- Jedno skladište*(engl. Store)*– Redux koristi jedno skladište unutar kojeg se nalaze sva stanja aplikacije. Iako se na prvu čini da držanje svih stanja unutar jednog skladišta može postati prekomplicirano i teško za skaliranje, ista stvar se dešava i sa upravljanjem prevelikog broja spremišta u različitim modulima. Uvođenjem jednog izvora stanja uvelike se pojednostavljuje razmišljanje o toku unutar aplikacije, predikciju nuspojava i otklanjanje grešaka koje uzrokuju akcije.
- Nepostojanje dispečera - akcije se šalju direktno u skladište bez potrebe za

dispečerom, tj. skladište i dispečer se smartaju istim konceptom.

- Reduktori(*engl. Reducers)* - uvode se funkcije koje se nazivaju reduktorima. Umjesto da se stanja domenski dijele unutar više spremišta kao što je to slučaj kod Flux arhitekture, u Reduxu ovu ulogu preuzimaju Reduktori. Reduktori su ustvari funkcije koje specificiraju kako transformirati stanja koja zaprime kada se otprema određena akcija, ove funkcije uzimaju trenutna stanja iz spremišta i odgovarajuću akciju te vraćaju ažurirana stanja.(Boduch, 2016)

Kako bi se redux mogao koristiti unutar React-a, potrebno je cijelu aplikaciju omotati react-redux *Provider* komponentom čime Redux skladište postaje dostupno kroz cijelo stablo komponenti aplikacije. U projektnom zadatku ovog rada korištena je Redux biblioteka u jednoj od implementacija, a na slici 12. je prikaz omatanja komponenti iz koda projektnog zadatka.

```
ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById("root")
);
```

*Slika 12 Prikaz omatanja aplikacije ReduxProvider komponentom*

Nakon što su sve komponente omotane *Provider* komponentom, moguće je pristupiti stanjima skladišta u bilo kojoj komponenti aplikacije, osim toga u Redux implementaciji projektnog zadatka korišten ja hooks API te se omatanjem omogućava korištenje react-redux hook funkcija u funkcionalnim komponentama. Jedna od takvih hook funkcija je *useSelector()* putem koje se na jednostavan način izvlače stanja iz Redux skladišta. Na slici 13. je prikazan primjer korištenja ove hook funkcije iz koda projektnog zadatka.

```
const info = useSelector((state) => state.userProfile.info);
```

*Slika 13 Prikaz korištenja useSelector() hook funkcije u kodu*

### 4.6.1 Redux akcije

Redux akcije su definirane i bazirane na principu prethodno obrađenih akcija Flux arhitekture te se obično sastoje od svojstva *type* koji predstavlja naziv akcije koja se izvodi i svojstva *value* koji predstavlja podatake koje se prosljeđuje skladištu. U projektnom zadatku uz react-redux *hooks* API je u komponentama omogućeno jednostavno otpremanje akcija. Funkciji *dispatch()* sa slike 14. pristupa se pomoću hook funkcije iz react-redux biblioteke pod nazivom *useDispatch()* koja vraća referencu na *dispatch()*iz skladišta. U slučaju sa slike 14. u dispatch funkciji šalje se prethodno uvezena *(engl. Imported)*akcija *updateProfile*.

```
dispatch(
  updateProfile({
    property: "availability",
    value: { [activeDay]: activeDayTime },
  })
```

*Slika 14 Prikaz sintakse Redux akcije unutar koda*

### 4.6.2 Redux skladište

Kako je već spomenuto, u Redux-u postoji samo jedno skladište stanja ili tzv. jedan izvor istine *(engl. Single Sourceoftruth).*Na razini koda, kreacija skladišta je vrlo jednostavna, na slici 15. prikazana  je implementacija u kodu iz projektnog zadatka. Redux funkcija za kreaciju skladišta pod nazivom *configureStore()* kao argument prima jedan reduktor koji može biti root reduktor ili reduktor koji kombinira više *slice* reduktora. Na primjeru iz projektnog zadatka korišteni su *slice* reduktori, a pri kreaciji skladišta je proslijeđen jedan *slice* reduktor.

```
export default configureStore({
  reducer: {
    userProfile: userProfileReducer,
  },
});
```

*Slika 15 Prikaz kreacije Redux skladišta unutar koda*

### 4.6.3 Redux reduktori

Reduktori su jedan od glavnih Redux koncepata, a tipična reduktor funkcija prilikom izvođenja prvo pregleda *type* svojstvo akcije kako bi identificirala kako odgovoriti, a nakon toga ažurira stanja tako da kreira kopije stanja koje je potrebno mijenjati i nakon toga mijenja ta stanja. U projektnom zadatku ovog rada korišteni su tzv. *slice*reduktori, a na slici 16. je prikaz definiranja slice reduktora putem Redux API-a *createSlice()* koji kao parametre prima naziv slice-a, inicijalno stanje te reduktor funkcije. Korištenjem *createSlice()* metode automatski se generiraju kreatori akcija te tipovi akcija koji odgovaraju proslijeđenim reduktorima i stanjima. Pa se tako npr. generirani tip akcije reduktor funkcije *updateProfile* sa slike 16. generira u tip *profile/updateProfile.* Osim toga treba naglasiti da se u ovom primjeru stanja mijenjaju „direktno“ što dozvoljava dodatna biblioteka pod nazivom immer.

```
export const userProfileSlice = createSlice({
  name: "profile",
  initialState: {
    selectedJobs: {},
    selectedLanguages: data.additionalLanguages.reduce(
      (res, item, idx) => Object.assign(res, { [item.name]: false }),
      {}
    ),
    availability: data.days,
    info: {
      fullName: "",
      city: "",
      date: "1999-01-01",
      genre: "F",
      phoneNumber: "",
    },
    credentials: {
      mail: "",
    },
  },
  reducers: {
    updateProfile: (state, action) => {
      const { property, value } = action.payload;
      state[property] = { ...state[property], ...value };
    },
  }})
```

*Slika 16 Prikaz sintakse reduktora unutar koda*

## 4.7 MobX

MobX je „reaktivna“ biblioteka otvorenog koda kreirana od strane Michael Westrate-a namjenjena za upravljanje stanja klijentskih aplikacij, a za razliku od Redux-a namijenjen je za manje aplikacije sa jednostvnijim modelom podataka. Za MobX se

kaže da je vrlo jednostavan za naučitit i koristiti jer je većina funkcionalnosti koje je potrebno explicitno definirati u drugim bibliotekama za upravljanje stanjem u MobX-u apstrahirano i radi automatski.

Za razliku od Redux stanja koja se definiraju kroz običnu strukturu podataka, stanja u MobX-u se definiraju tzv. *observable* strukturom podataka, osim toga stanja u MobX-u nisu nepromjenjiva tj. moguće ih je mijenjati „direktno“ ili kroz akcije. Kada se stanja unutar MobX-a promjene, aplikacija reagira na te promjene kroz tzv. reakcije *(eng. Reactions),* a primjer tih reakcija može biti jednostavno ažuriranje *view* razine. Gledajući tok u MobX-u, između observable MobX stanja te MobX reakcija moguće je dodati još jedan sloj, a to su tzv.izračunate vrijednosti (*engl. CalculatedValues*) koje u MobX-u nisu obavezne za korištenje ali mogu biti od velikog značaja pogotovo u slučaju da se želi zadržati jednostavna struktura stanja u skladištima. Izračunate vrijednosti su ustvari izvedena svojstva iz stanja ili iz drugih izračunatih vrijednosti, a ne ažuriraju se svaki put kada se stanja mijenjaju nego samo kada se koriste u reakcijama koje ažuriraju *view* sloj(Podila, Westrate, 2018).

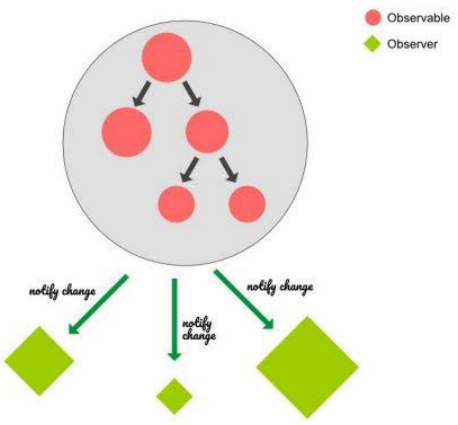
### 4.7.1 Observable stanja

MobX pruža funkcionalnost pod nazivom *observable* putem koje se definiraju „reaktivna“ stanja aplikacije. Bilo koji JavaScript objekt može postati *obesrvable,* a na slici 17. prikaz je definiranja observable stanja u kodu projektnog zadatka sa MobX implementacijom. Koristila se *makeAutoObservable()* MobX funkcija koja u konstruktoru klase kao parametar prima instancu klase kroz *this* i pretvara sva svojstva klase u observable stanja, a sve funkcije klase kao npr. *setProfile()* u akcije. Prilikom kreacije observable stanja moguće je i eksplicitno definirati tip svakog člana klase.

```
You, seconds ago | 1 author (You)
class UserProfileStore {
  selectedJobs = {};
  selectedLanguages = [];
  availability = data.days;
  info = {
    fullName: "",
    city: "",
    date: "1999-01-01",
    genre: "F",
    phoneNumber: "",
  };
  credentials = {
    mail: "",
  };
  setProfile(value, property) {
    this[property] = { ...this[property], ...value };
  }
  constructor() {
    makeAutoObservable(this);
  }
}
```

*Slika 17 . Prikaz kreacije observable stanja u MobX-u*

*Observable* stanja, prikazana kao krugovi na slici 18. su stanja koje je unutar aplikacije moguće „promatrati“. Svaki put kada se unutar njih dogodi promjena, *observer* komponente, prikazane kao rombovi unutar dijagrama bivaju „obaviještene“. Observer je u stanju „promatrati“ jedan ili više *observable* stanja te biva „obaviješten“ kada god jedan od njih promijeni unutarnju vrijednost(Podila, Weststrate, 2018).

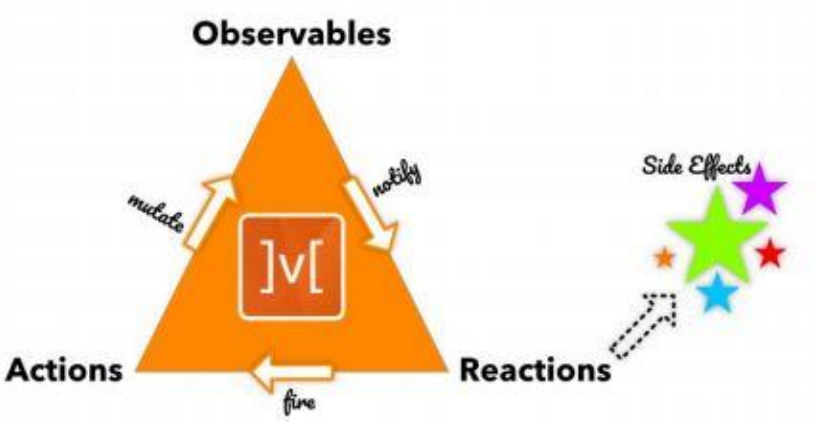



*Slika 18 Prikaz međuodnosa observer i observable komponenti*

Stanja u MobX se u pravilu ne drže u jednom globalnom objektu nego se mogu nalaziti u više skladišta stanja *(engl. State Store).*
Kako bi se omogućilo reagiranje na promjene *observable* stanja potrebno ih je „promatrati“.

### 4.7.2 Reakcije

Reakcije *(engl. Reactions)* su tzv. nuspojave koje uzrokuju određena ponašanja koja se izvršavaju prilikom  promjene unutar *observables* stanja na koje su pretplaćeni. Reakcije imaju ulogu „promatranja“ observables stanja i „reagiranje“ na njih kao što je i prikazano na sljedećem dijagramu.




*Slika 19 Prikaz toka događaja unutar MobX-a*

U MobX-u postoji više vrsta reakcija, a to su:(Podila, Westrate, 2018):

- *autorun()* je tip reakcije koja kao argument uzima funkciju unutar koje se izvršavaju nuspojave *(engl.Side-effect)*, te nuspojave mogu ovisiti o jednom ili više observable stanja. MobX *autorun()* se izvršava prilikom inicijalne interpretacije te prilikom svake promjene na *observable* stanjima o kojima su nuspojave unutar *autorun()* ovisne automatski poziva funkciju kako bi se te nuspojave izvršile. Stanja o kojima su ovisne nije potrebno explicitno definirati već se u MobX-u to obavlja automatski.
- *reaction()* je tip reakcije sličan *autoran()* reakciji sa dodatnom mogućnošću odvajanja zadaća praćenja promjena koje se događaju unutar *observable* stanja i reagiranja na njih. Ovime se dobija mogućnost odlučivanja o tome

kada se nuspojave izvršavaju za razliku od *autorun()* reakcije gdje se nuspojave uvjek izvršavaju pri mjenjenju stanja o kojima ovisi.

- *when()* je tip reakcije gdje se nuspojave izvrše samo jednom nakon što se ispune zadani uvjeti i nakon toga se u *when()* reakciji promjene stanja više ne promatraj u i samim time njene nuspojave više ne izvršavaju

### 4.7.3 Akcije

U MobX-u je stanja moguće mijenjati direktno, međutim na ovaj način mijenjanja stanja se često gleda kao loša praksa. Direktno mijenjanje unutar komponenti također usko veže *(engl. Coupling )*promjene stanja sa *view* slojem. Korištenjem MobX akcija mijenjanje stanja se odvaja*(engl.Decoupling*) od *view* sloja te se sve funkcije mijenjanja stanja nalaze na jednom mijestu, unutar akcija.(Wieruch, 2017) Akcije skrivaju detalje o tome kako se promjene moraju izvršiti ili na koja *observable* stanja se treba utjecati. Akcije pridonose razumijevanju koda pridavajući deklarativna imena operacijama koje mijenjaju stanja.

## 4.8 Recoil

Recoil je biblioteka otvorenog koda za upravljanje stanjima kreirana od strane Facebook-ovog programera Dave McCabe-a. Za razliku od Redux-a, a donekle i MobX-a, Recoil predstavlja vrlo jednostavan API te ne zahtjeva puno ponavljajućeg koda i konfiguracije prije nego što se može početi koristiti. Recoil je nastao kao rezultat  mnogih problema na koji je Facebook-ov tim naišao prilikom rada na internim projektima, a zbog kompleksnosti tih projekata konvencionalne biblioteke za upravljanje stanja im ili nisu bile od pomoći ili je implementacija istih bila pre dugotrajna. Recoil kao riješenje nudi riješenje koje je sintaksno jako slično React-u, a konceptualno je vrlo jednostavno za razumijeti.Tok promjene stanja u Recoil-u kreće od tzv. Recoil atom-a unutar kojih se drže stanja, prolazi kroz tzv. Recoil *selector* koji je definiran kao pure funkcija i završava u React komponenti.

Recoil donosi niz dodataka React-u, a to su:

- Globalno dijeljenje stanja – dijeljenje istih stanja različitim komponentama u

React hijerarhijskom stablu na vrlo optimiziran i prediktivan način

- Izvođenje podataka i upiti – izvođenje podataka iz stanja bazirano na promjenama unutar tih stanja, a izvođenja mogu biti sinhrona ili asinhrona
- „Promatranje“ stanja kroz cijelu aplikaciju – „promatranje“ promjena, ispravljanje pogrešaka kroz vrijeme i mnoštvo drugih mogućnosti

### 4.8.1 Atomi

Recoil atomi predstavljaju dijelove ili jedinice stanja. Stanja unutar atoma je moguće ažurirati te je moguće pretplatiti se na njih. Kada se atom ažurira, svaka pretplaćena komponenta se ažurira sa novim stanjima. Ukoliko se isti atom koristi u više React komponenti, svaka od tih komponenti dijeli stanja tog atoma. Atomi se u kodu kreiraju na sljedeći način:

```
export const credentialsState = atom({
  key: "credState",
  default: {
    mail: "",
    password: "",
  },
});
```

*Slika 20. Prikaz kreacije atoma u kodu unutar Recoil-a*

Atomu je potrebno zadati unikatno *key* svojstvo te početnu vrijednost stanja u atomu pod *default* koja je u slučaju sa slike 20. u obliku objekta ali može biti u bilo kojem obliku kao što je lista, funkcija, JavaScript primitiv i dr.
Kako bi se komponenta pretplatila na atom, koristi se *useRecoilState* unutar kojeg kao argumant postavljamo atom na koji se želimo pretplatiti. Ovaj hook je vrlo sličan *setState* hook-u te opskrbljava komponentu sa vrijednošću stanja tog atoma te funkcijom za ažuriranje stanja. Primjer korištenja *useRecoilState* iz projektnog zadatka nalazi se na slici 21.

```
export default function CredentialsStep(props) {
  const [visible, setVisible] = useState(false);
  const [info, setInfo] = useRecoilState(credentialsState);
```

*Slika 21 Prikaz pretplate na atom u kodu*

### 4.8.1 Selektori

Selektori su *pure* funkcije koje primaju atome ili druge selektore kao ulaz, kada se ulazni selektori ili atomi ažuriraju, selektor se ponovno izvršava. Komponente se osim na atome mogu pretplatiti i na selektore te se ponovno ažuriraju kada se i selektor na koji su pretplaćeni mijenja.
Selektori se koriste za izvođenje (engl. Derive) podataka baziranih na stanjima, a to nam omogućava izbjegavanje bespotrebno velikih i kompleksnih stanja u atomima. Selektori prate koje komponente su pretplaćene na njih i o kojim su stanjima u atomima ovisni. Sa gledišta komponenti, selektori i atomi imaju identično sučelje te su lako zamjenjivi. Selektori se u kodu kreiraju na sljedeći način:

```
const checkLoadingState = selector({
  key: "loadingState",
  get: ({ get }) => {
    const loading = get(loadingState);
    return "Loading is" + loading ? "true" : "false";
  },
});
```

*Slika 22 Prikaz kreiranja selektora u kodu*

Na slici 22. *get* svojstvo je funkcija za izvođenje podataka, koristeći *get* koji prima kao argument pristupa stanjima u atomu *loadingState*. Prilikom pristupanja stanjima tog atoma, kreira se veza gdje ažuriranje tog atoma uzrokuje selektor na ponovno izvođenje.  Vrijednostima selektora je unutar komponente moguće pristupiti korištenjem *useRecoilValue* koji kao argument uzima selektor te vraća odgovarajuću vrijednost, primjer pristupanja selektoru prikazan je na slici 23.

```
const fontSizeLabel = useRecoilValue(fontSizeLabelState);
```

*Slika 23 Prikaz pristupanja selektoru iz komponente u kodu*

# 5. Projektni zadatak

Za demonstraciju koncepata iz ovog rada napravljen je projektni zadatak u obliku klijentske web-aplikacije korištenjem biblioteke React, projekt je kreiran pomoću *create-react-app* razvojnog okruženja koje omogućava korištenje najnovijih JavaScript značajki te postavlja i konfigurira projekt po najboljoj praksi*(engl. Best practices).*
Sav izvorišni kod praktičnog dijela i sve implementacije po granama dostupne su na sljedećoj GitHub poveznici: https://github.com/tomiandic/firmus.

## 5.1 Uvod u izradu projektnog zadatka

Za definiranje izgleda i propagiranje teme kroz aplikaciju korištena je Material UI biblioteka. Projektni zadatak je napravljen u više implementacija ovisno o vrsti i biblioteci korištenoj za upravljanje stanjima, a za upravljanje stanja korišteni su:

- Lokalna stanja
- Context API
- Redux
- MobX
- Recoil

## 5.2 Proces izrade

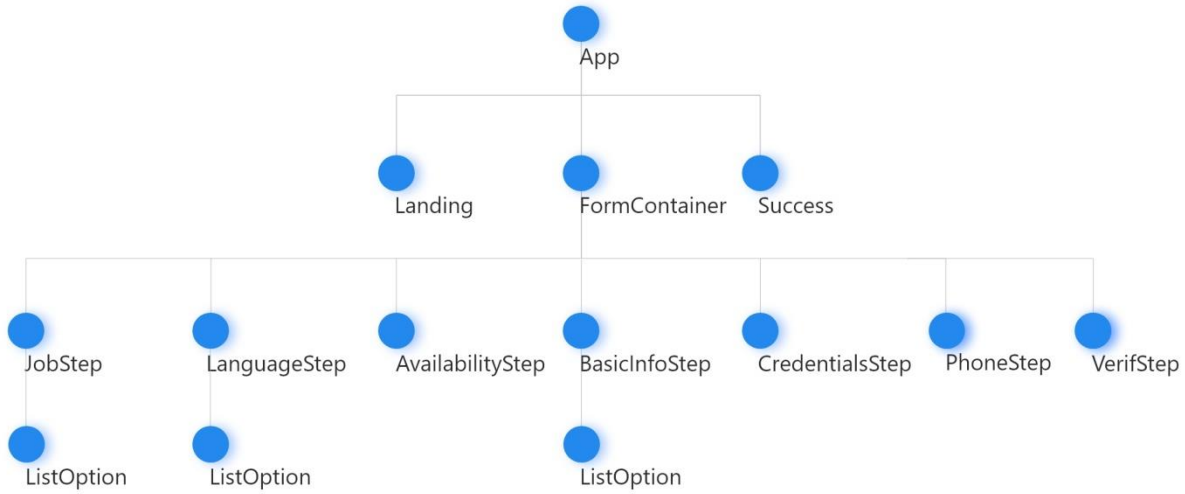
Prije početka razvoja aplikacije u kodu napravljen je grafički prototip putem kojeg se kroz više iteracija analize izkristalizirao I definirao izgled aplikacije I tok korisničkog iskustva. Tek nakon što je prototip konačno definiran kreće se sa implementacijom u kodu što je uvelike umanjilo vrijeme I nepotrebnu konfuziju.
Prije početka implementacije su također istražene najpopularnije biblioteke za propagaciju teme I biblioteke potrebne za ostvarivanje željenih interakcija sa korisnikom uz konstantno referiranje na prototip.

Nakon toga je definirana arhitektura direktorija unutar projekta te arhitektura React hijerarhijskog stable projekta te se definirao JSON koji oponaša bazu podataka. Prva implementacija izvršila se koristeći lokalna stanja, a nakon toga je iz te kodne baze

izvučeno više git grana, a u svakoj od tih grana je kod refaktoriran sa implementacijom jedne od nabrojanih biblioteka za upravljanje stanjima. Prilikom implementacije aplikacije u kodu korišten je tzv. mobile-first pristup, a sama aplikacija je responzivna te podržava većinu rezolucija uređaja.

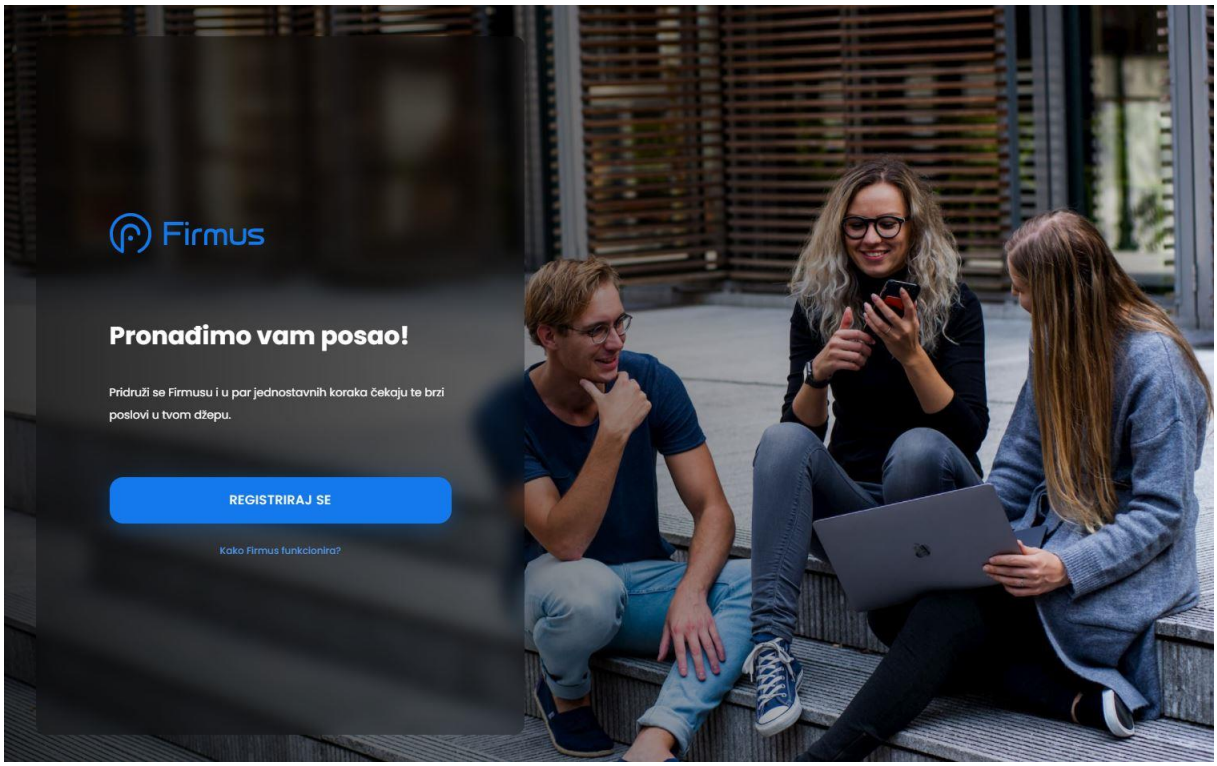
## 5.3 Aplikacija za registraciju studenata na servis za zapošljavanje „Firmus“

Za projektni zadatak napravljena je klijentska aplikacija čija je glavna funkcionalnost registracija studenata u svrhu zapošljavanja. Aplikacija se sastoji od komponente za prikaz početne naslovne stranice, komponente koja upravlja koracima registracije, a svaki od koraka je zasebna komponenta. Struktura aplikacije gdje svaka komponenta predstavlja čvor unutar stabla prilazana je na slici 24.



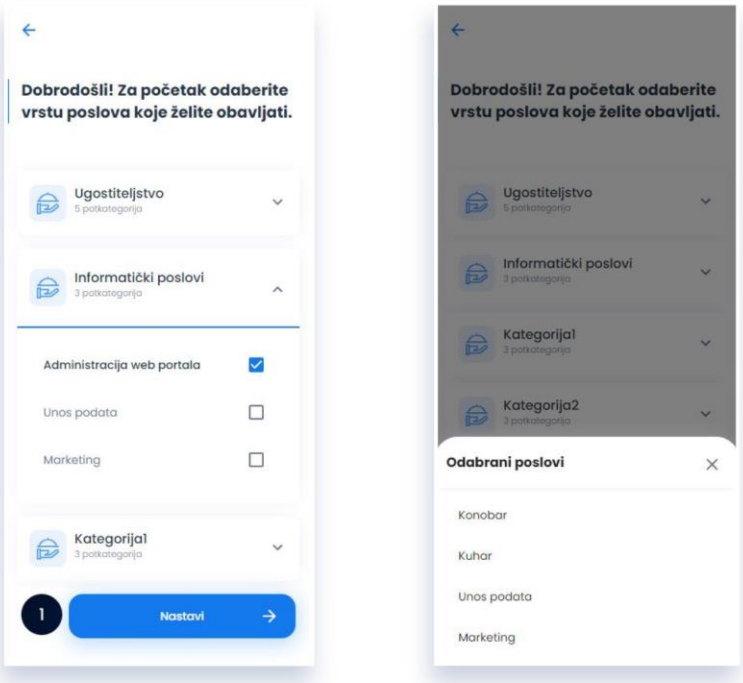

*Slika 24 Prikaz kompozicije aplikacije u obliku stabla*

Na naslovnoj stranici aplikacije prikazanoj na slici 25. korisnik ima opciju početka registracije ili odabira prikaza instrukcije o radu servisa „Firmus“. Odabirom gumba za instrukcije korisniku su prikazane informacije o načinu rada servisa kroz korake. Odabirom gumba za registraciju korisnik je preusmjeren na prvi korak registracijske forme.




*Slika 25 Prikaz početne stranice aplilkacije*

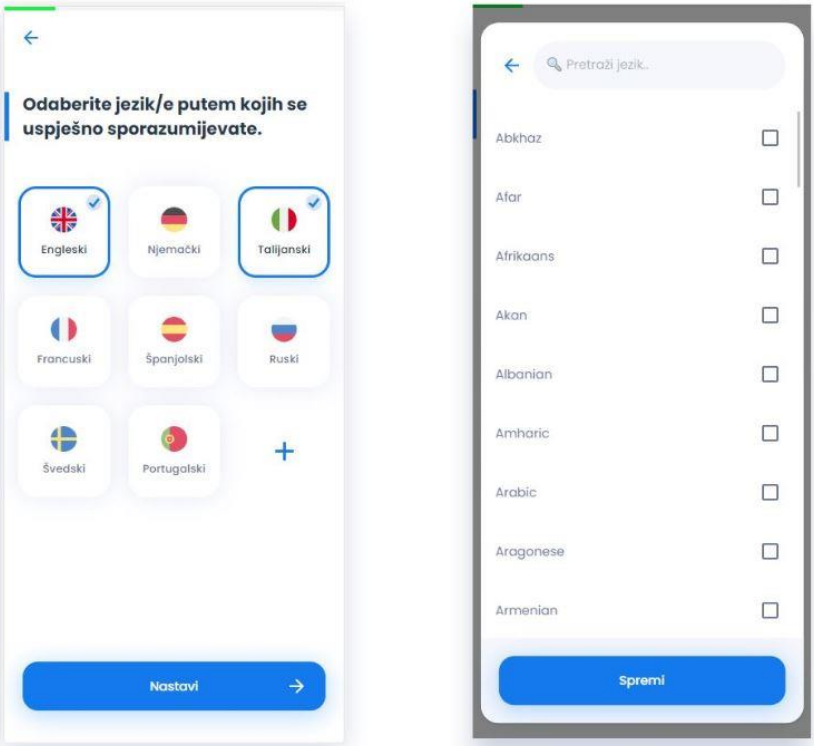
Na prvom koraku registracijske forme prikazanoj na slici 26. korisniku je prikazana lista kategorija poslova (u obliku komponenata Accordion iz Material UI biblioteke), klikom na pojedinu kategoriju korisniku se prikazuje lista poslova koji spadaju u odabranu kategoriju. Odabirom barem jednog od poslova korisniku se prikazuje gumb „nastavi“ te gumb sa brojem odabranih poslova, klikom na gumb sa brojem odabranih poslova korisniku se prikazuje lista svih odabranih poslovakao što je prikazano na drugom ekranu sa slike 26., klikom na gumb „nastavi“ korisniku se prikazuje drugi korak registracije.




*Slika 26 Prikaz prvog koraka registracije*

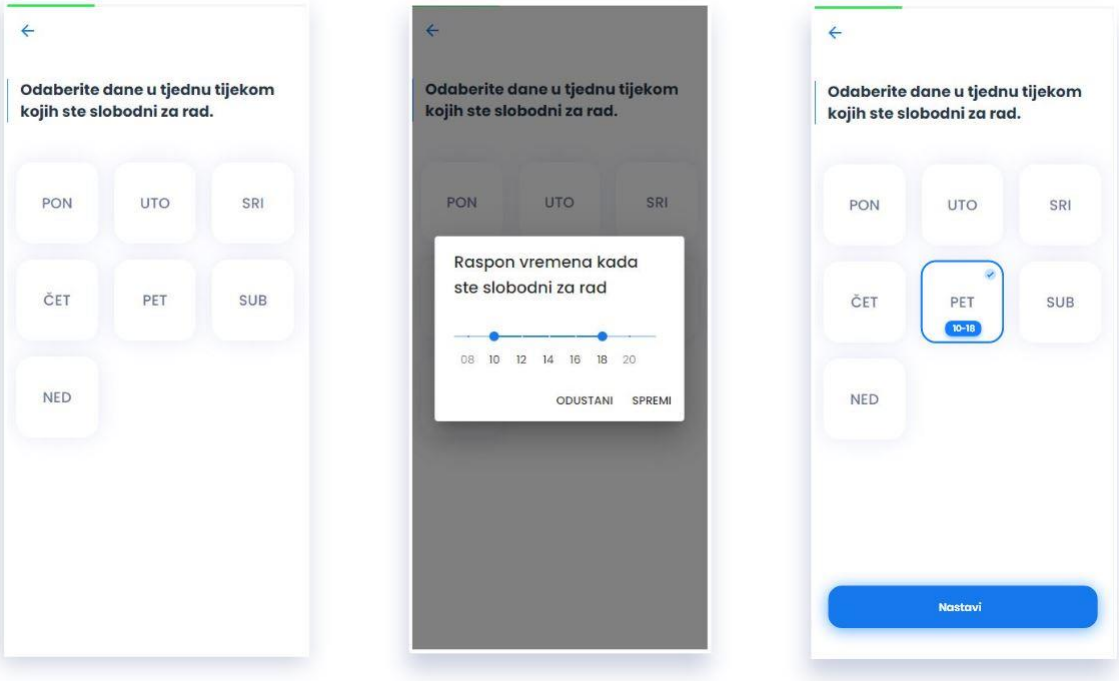
Na drugom koraku registracijske forme prikazanom na slici 27. korisniku je prikazana lista jezika u obliku kartica, korisnik je na ovom koraku u mogućnosti odabrati jedan ili više jezika koje poznaje. Ukoliko korisnik poznaje jezik koji nije prikazan putem kartice, klikom na gumb sa znakom plus korisniku se prikazuje modalni prozor sa listom svih jezika prikazan na drugom ekranu slike 27. U modalnom prozoru korisnik je u mogućnosti pretraživati željeni jezik te odabrati jedan ili više jezika. Kako bi se optimizirao rad prilikom prikaza preko 150 opcija jezika gdje svaka opcija definirana putem jedne komponente, koristila se biblioteka za virtualizaciju pod nazivom react-virtualized.

Korisnik prelazi na sljedeći korak registracije putem pritiska na gumb „nastavi“.



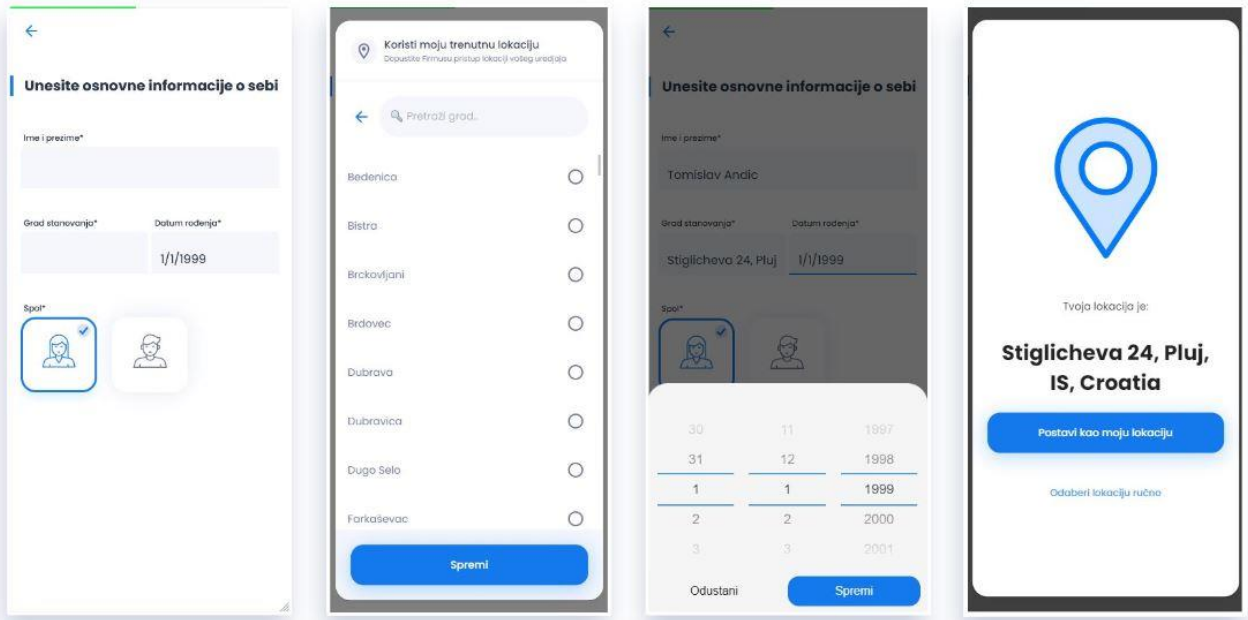

*Slika 27 Prikaz drugog koraka registracije*

Na trećem koraku registracijske forme prikazanom na slici 28. korisniku je prikazana lista dana u tjednu u obliku kartica. Korisnik je u ovom koraku u mogućnosti odabrati vrijeme u tjednu kada je slobodan za rad, klikom na jednu od kartica korisniku se prikazuje modalni prozor za odabir vremena u danu. Nakon što korisnik specificira raspon vremena kada je slobodan u odabranom danu, klikom na gumb “spremi” korisniku se prikazuje označena kartica sa odabranim rasponom vremenate se prikazuje gumb “nastavi” za prelazak na sljedeći korak registracije. Korisnik mora odabrati barem jedan raspon vremena kako bi mogao nastaviti registraciju.



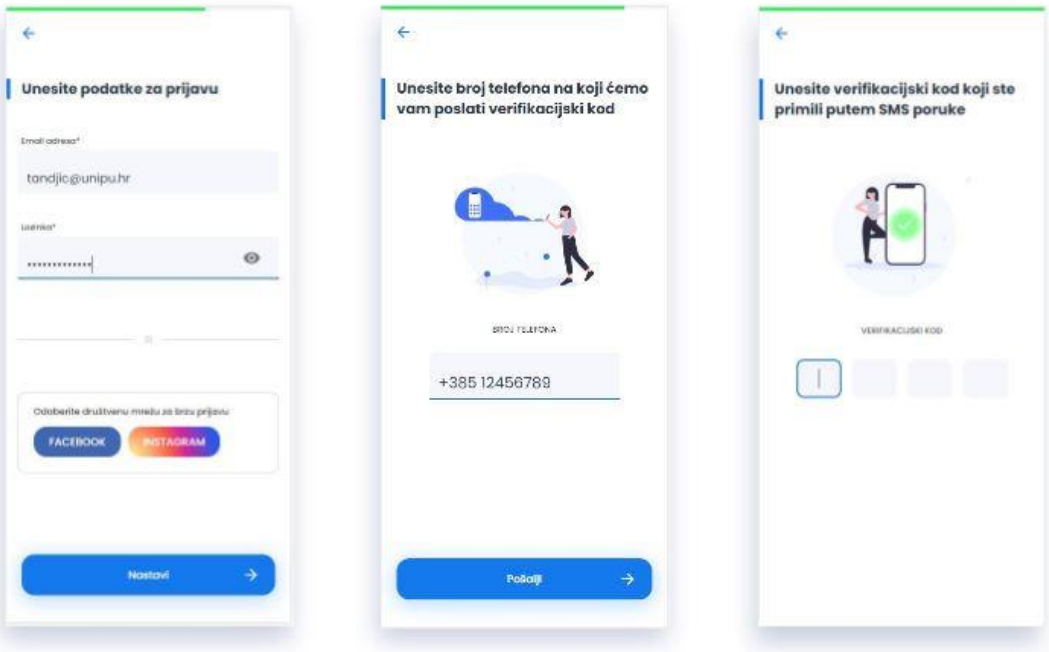

*Slika 28 Prikaz trećeg koraka registracije*

Na četvrtom koraku registracijske forme prikazanom na slici 29. korisniku se prikazuje forma za unos osnovnih informacija. Klikom na dio forme za unos grada stanovanja otvara se modalni prozor prikazan na drugom ekranu slike 29. Gdje korisnik može pretražiti I odabrati svoj grad stanovanja, osim toga korisnik ima I opciju za automatsku detekciju grada stanovanja putem lokacije uređaja. Klikom na gumb “Koristi moju trenutnu lokaciju” korisniku se ispisuje njegova lokacija kao što je prikazano načetvrtom ekranu slike 29. Za dobijanje lokacije koristio se JavaScript Navigator API za pristupanje lokacijskim informacijama uređaja te API Positionstack servisa koji za parametre prima zemljopisnu dužinu*(engl. Longitude)* I širinu*(engl. Latitude)* te vraća detalje o lokaciji i adresu.
Ukoliko korisnik odluči da je identificirana lokacija točna, klikom na “postavi kao moju lokaciju”, lokacija se sprema te se modalni prozor zatvara.Klikom na dio forme za unos datuma rođenja otvara se modalni prozor prikazan natrećem ekranu slike 29. gdje korisnik odabire datum rođenja. Nakon što se sva polja ispune, korisniku se prikazuje gumb za nastavak.




*Slika 29 Prikaz četvrtog koraka registracije*

Zadnja tri koraka registracijske forme prikazana su na slici 30. u prvom koraku sa slike korisniku se prikazuje forma za unos detalja korisničkog računa, nakon klika na gumb “nastavi” korisniku se prikazuje drugi korak sa slike gdje je forma za unos broja telefona, korisnik mora upisati minimalno devet brojeva kako bi mu se prikazao gumb za slanje. Nakon klika na gumb “pošalji” korisniku se prikazuje treći korak sa slike gdje je predviđena verifikacija putem verifikacijskog koda, verifikacija putem SMS poruke nije implementirana u ovom projektnom zadatku te korisnik završava registraciju unosom bilo koja četiri znaka na zadnjem koraku. Nakon što se unes u četiri znaka korisnik se automatski usmjerava na zadnji pogled gdje mu se prikazuje pregled registriranog profila.




*Slika 30 Prikaz zadnja tri koraka registracije*

Na zadnjem pogledu aplikacije korisniku je prikazan ispis svih detalja profila koji je kreirao putem registracijskih koraka.

*Slika 31 Prikaz profila na zadnjem dijelu aplikacije*

# 6.Zaključak

U ovom radu obrađena je problematika upravljanja stanjima unutar React programske biblioteke. Rad se sastoji od teorijskog dijela i praktičnog dijela. U teorijskom dijelu postoji kratki pregled JavaScript programskog jezika, načina njegovog izvođenja i pisanja, nakon toga je obrađena sama biblioteka React i svi njeni glavni dijelovi. U četvrtom poglavlju je obrađen koncept uptravljanja stanja te je prikazan niz načina kako ostvariti upravljanje stanja u React biblioteci. U petom poglavlju je prikazan praktični dio te sve funkcionalnosti aplikacije.

Kroz projektni zadatak iz praktičnog dijela koji se sastoji od više implementacija korištenjem različitih biblioteka za upravljanje stanjem prikazana su trenutno najkorištenija i najpopularnija riješenja poput Redux-a kao i neka novija rješenja kao što je Recoil.

Kroz pregled ovog rada može se zaključitit kako se način upravljanja stanja u aplikaciji odabire ovisno o faktorima kao što su kompleksnost aplikacije koja se gradi, njena kompozicija komponenti i način planiranog međudjelovanja komponenti. Iz ovog razloga vrlo je važno planiranje aplikacije unaprijed kako bi se povećala efikasnost procesa izrade aplikacije.

Kroz pregled svih implementacija praktičnog dijela može se zaključiti da koristeći jednostavnije biblioteke gdje je potrebno manje koda i gdje se nudi više apstrahiranih funkcija uglavnom se žrtvuje na elastičnost i mogućnost skalabilnosti, dok složenije biblioteke nude bolje odvajanje i neovisnost različitih slojeva u sustava te veću razinu kontrole i skalabilnost.

# Literatura


1. Wieruch R. (2017), Tamingthe State in React:Your journey to master Redux and MobX. Lean Publishing

2. Ford N. (2014), Functional Thinking: Paradigm over Syntax. USA:O'Reilly Media, Inc

3. Sidelnikov G. (2016), React.js Book: Learning React JavaScript Library From Scratch. River Tigris LLC

4. Garreau M. iFaurot W. (2018), Redux in Action, Manning

5. Flanagan D. (2020), JavaScript: The Definitive Guide, Seventh Edition, USA:O'Reilly Media, Inc

6. Boduch A. (2016), FluxArchitecture: Learn to build powerful and scalable applications with Flux, the architecture that serves billions of Facebook users everyday. PacktPublishingLtd.

7. Podila P. i Westrate M.(2018), MobXQuick Start Guide: Supercharge the client state in your React apps with MobX, PacktPublishingLtd.

8. Context API documentation [Web izvor], <raspoloživo na: https://reactjs.org/docs/context.html>