Usporedba okvira za razvoj mobilnih aplikacija

Kumir, Marko

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

ZAVRŠNI RAD

USPOREDBA OKVIRA ZA RAZVOJ MOBILNIH APLIKACIJA

Marko Kumir

Split, rujan 2022.

Temeljna dokumentacijska kartica

Završni rad

Sveučilište u Splitu
Prirodoslovno-matematički fakultet
Odjel za informatiku
Ruđera Boškovića 33, 21000 Split, Hrvatska

USPOREDBA OKVIRA ZA RAZVOJ MOBILNIH APLIKACIJA

Marko Kumir

SAŽETAK

Ovaj završni rad pojašnjava pojam i koncepte reaktivnog programiranja. Opisani su popularni okviri koji koriste navedene koncepte pri izvedbi. U praktičnom dijelu rada implementirana je mobilna aplikacija u dva različita okvira. Okviri React Native i Svelte Native su uspoređeni na temelju njihove strukture, povezivanja podataka, navigacije i upravljanja stanjem.

Ključne riječi: okvir, mobilna aplikacija, reaktivno programiranje, JavaScript, React Native, Svelte Native

Rad je pohranjen u knjižnici Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu

Rad sadrži: 33 stranice, 42 grafička prikaza, 2 tablice i 12 literaturnih navoda. Izvornik je na hrvatskom jeziku.

Mentor: **Dr. sc. Divna Krpan,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Ocjenjivači: **Dr. sc. Divna Krpan,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Dr. sc. Saša Mladenović, *izvanredni profesor Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Dr. sc. Goran Zaharija, *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Rad prihvaćen: rujan 2022.

Basic documentation card

Thesis

University of Split
Faculty of Science
Department of Computer Science
Ruđera Boškovića 33, 21000 Split, Croatia

COMPARISON OF MOBILE APPLICATION DEVELOPMENT FRAMEWORKS

Marko Kumir

ABSTRACT

This thesis clarifies the term and concepts of reactive programming. Popular frameworks that use the above concepts in performance are described. In the practical part of the thesis, a mobile application was implemented in two different frameworks. React Native and Svelte Native were compared based on their structure, data binding, navigation and state management.

Key words: framework, mobile application, reactive programming, JavaScript, React Native, Svelte Native

Thesis deposited in library of Faculty of science, University of Split

Thesis consists of:	33 pages, 42 figures, 2 tables and 12 references
	Original language: Croatian
Mentor:	**Divna Krpan, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*
Reviewers:	**Divna Krpan, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*
	Saša Mladenović, Ph.D. *Associate Professor of Faculty of Science, University of Split*
	Goran Zaharija, Ph.D. *Assistant Professor of Faculty of Science, University of Split*

Thesis accepted: September, 2022.

IZJAVA

kojom izjavljujem s punom materijalnom i moralnom odgovornošću da sam završni rad s naslovom USPOREDBA OKVIRA ZA RAZVOJ MOBILNIH APLIKACIJA izradio samostalno pod voditeljstvom dr.sc. Divne Krpan. U radu sam primijenio metodologiju znanstvenoistraživačkog rada i koristio literaturu koja je navedena na kraju završnog rada. Tuđe spoznaje, stavove, zaključke, teorije i zakonitosti koje sam izravno ili parafrazirajući naveo u završnom radu na uobičajen, standardan način citirao sam i povezao s fusnotama s korištenim bibliografskim jedinicama. Rad je pisan u duhu hrvatskog jezika.

Student

Marko Kumir

Sadržaj

Uvod

Iz dana u dan, broj mobilnih uređaja se sve više povećava, što dovodi do razvoja mnoštva inovativnih mobilnih aplikacija. Trenutno, tržište mobilnih aplikacija najbrže je rastući segment mobilne industrije. Također, danas je gotovo nemoguće zamisliti život bez njihove upotrebe.

Pri razvoju mobilnih aplikacija, programeri koriste dva glavna pristupa, a to su nativni pristup i višeplatformski pristup. Nativni pristup podrazumijeva razvoj aplikacije za određeni mobilni operacijski sustav, dok višeplatformski pristup omogućava izvođenje aplikacije na više mobilnih operacijskih sustava, od kojih su najpopularniji Android i iOS. U kontekstu naslova rada, pojam okvir označava biblioteku koja pruža osnovnu strukturu potrebnu pri razvoju mobilne aplikacije za određeno okruženje. Razlog korištenja okvira je zapravo ušteda na vremenu prilikom izrade aplikacija. To se postiže korištenjem unaprijed kreiranih komponenata i funkcija koje nudi određeni okvir.

U teoretskom dijelu rada je opisana paradigma uz koju su pojašnjeni koncepti reaktivnog programiranja, te su navedeni najpopularniji okviri koji koriste opisane koncepte pri izradi aplikacija.

U praktičnom dijelu rada, implementirana je mobilna aplikacija u dva različita okvira. Primarni cilj ovog rada je usporediti strukturu i funkcionalnosti mobilnih aplikacija, realiziranih u React Native-u i Svelte Native-u.

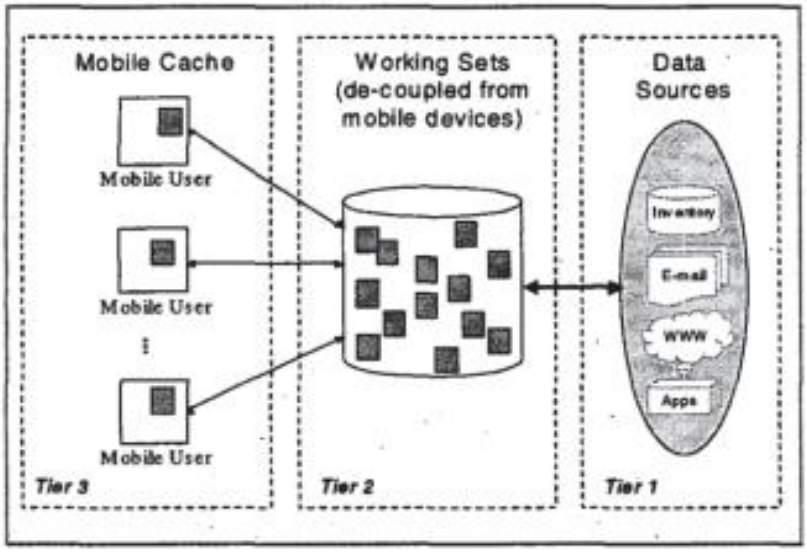
1. Mobilne aplikacije

Višeplatformske (engl. *Cross-platform*) mobilne aplikacije se, na temelju implementacije, mogu klasificirati u četiri različite kategorije. U kategorije spadaju:

- Mrežne aplikacije (engl. Web apps)
- Hibridne aplikacije
- Interpretirane aplikacije
- Aplikacije temeljene na kontrolama (engl. *Widget Based Apps*)

1.1. Mrežne aplikacije

Pod kategorijom mrežne aplikacije se nalaze aplikacije koje za pokretanje koriste internetski preglednik i imaju ograničen pristup sklopovlju mobilnog uređaja [1]. Glavne tehnologije koje se koriste za razvoj ovakvih vrsta aplikacija su HTML i JavaScript. Obično su popraćene okvirima poput Angular-a, jQuery-a, Django-a, Ruby on Rails-a itd. Mrežne aplikacije ne koriste izravno radnu memoriju uređaja. Sadržaj ovog tipa aplikacije je prikazan u pregledniku samog uređaja. Iz tog razloga, ova vrsta aplikacije ne zahtijeva instalaciju. Mrežne aplikacije slijede troslojnu arhitekturu. Prvi sloj je tanki sloj, odnosno sloj uređaja, drugi sloj je sloj radne aplikacije i treći sloj je sloj baze podataka.



Slika 1 Troslojna arhitektura mrežne mobilne aplikacije [1]

Sa stajališta izvedbe, mrežna aplikacija uključuje poslužitelj na kojem se nalaze datoteke, moguću instalaciju lokalnog resursa, deklaraciju metapodataka za proces instalacije, mehanizam za mrežno renderiranje i mehanizam za izvršavanje JavaScript-a koji učitava sadržaj [2]. S obzirom da je za dohvat podataka potreban poslužitelj, korisnici prilikom korištenja ovakve aplikacije moraju uvijek imati aktivnu internetsku vezu. Dostupne platforme za mrežne aplikacije su aplikacije na početnom zaslonu iPhone operacijskog sustava (engl. *iOS*) i progresivne mrežne aplikacije. Web aplikacije predstavljaju način instaliranja aplikacija, bez nativnih komplikacija i problema distribucije spremnika, stopostotno baziranih na mrežnom sadržaju.

1.2. Hibridne aplikacije

Karakteristika koja definira hibridne aplikacije je kombinacija nativnih dijelova s uobičajenim Internet tehnologijama kao što su HTML, CSS i JavaScript. Definicija, koja najpreciznije opisuje pojam hibridnih aplikacija, je mješavina nativnih i mrežnih aplikacija. Hibridne aplikacije su napisane u petoj verziji HTML-a poput mrežne aplikacije, ali su također instalirane i mogu pristupiti sklopovlju (engl. *Hardware*) uređaja poput nativnih aplikacija [3]. Drugim riječima, to su mrežne aplikacije smještene unutar nativne aplikacije koja koristi mrežni prikaz (engl. *WebView*) mobilne platforme za pokretanje i obradu JavaScript-a. Pojam mrežni prikaz označava preglednik, koji je u paketu s mobilnom aplikacijom.

Hibridne mobilne aplikacije izgrađene su na sličan način kao i mrežne stranice. Obe vrste aplikacija koriste kombinaciju istih tehnologija. Međutim, umjesto ciljanja na mobilni preglednik, hibridne aplikacije ciljaju na mrežni prikaz koji je smješten unutar nativnog spremnika. To im zapravo omogućuje pristup funkcionalnostima sklopovlja mobilnog uređaja. Implementacija koda hibridnih aplikacija može se izvršiti pomoću raznih tehnologija i razvojnih platformi, ali kako bi se postigao nativni dojam i izgled aplikacije, potrebno je koristiti specifične biblioteke kao što je npr. JQuery.

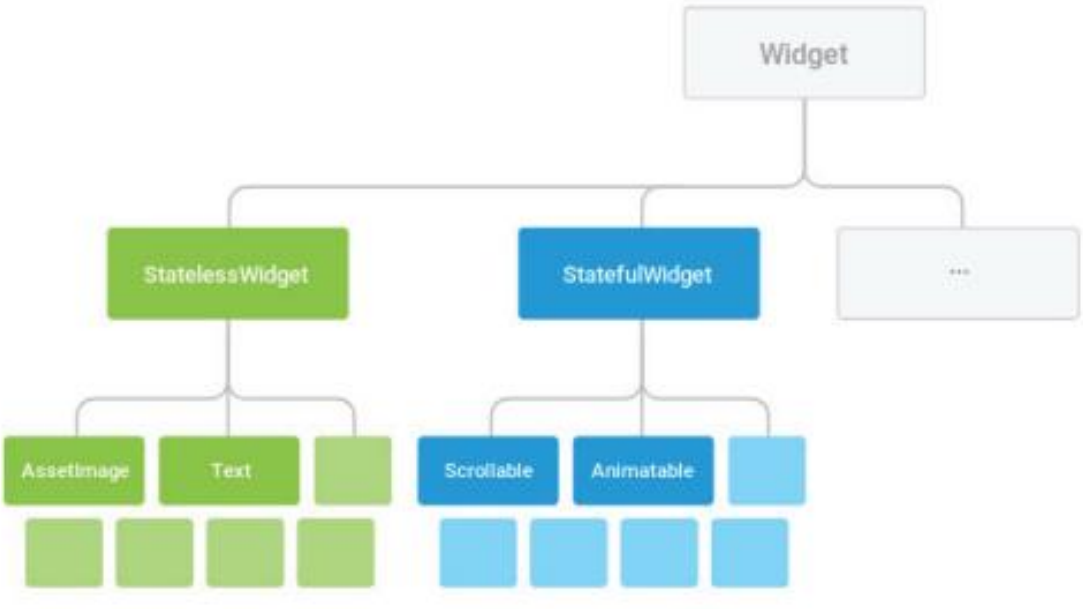
1.3. Interpretirane aplikacije

Interpretirana aplikacija je aplikacija čijoj je korisničko sučelje u nativnom kodu koji se interpretira pomoću zajedničkog jezika za različite operacijske sustave. Krajnji korisnici komuniciraju s komponentama nativnog korisničkog sučelja specifične platforme, dok se

logika aplikacije neovisno implementira koristeći nekoliko tehnologija i jezika, kao što su Java, Ruby, XML itd. [4]. Glavna prednost ovog pristupa je učinkovitost zbog nativnih korisničkih sučelja. Nedostatak je potpuna ovisnost o razvojnom okruženju za izradu programske podrške (engl. *Software Development Environment*, skraćeno SDE). Preciznije rečeno, nove značajke specifične za platformu mogu biti dostupne aplikacijama samo, kada, i ako ih podržava razvojno okruženje.

1.4. Aplikacije temeljene na kontrolama

Aplikacijske kontrole su svi dijelovi aplikacije koji se mogu definirati kao strukturni elementi, poput padajućeg izbornika, gumba, itd. Pod kontrole spada i stilski element, kao što je tema ili boja fonta, ili značajka izgleda poput margine . Sve pripadajuće komponente aplikacije služe kao građevni blokovi za oblikovanje korisničkog sučelja aplikacije. Aplikacija temeljena na kontrolama (engl. *Widget Based App*) tretira svaku komponentu kao kontrolu i na taj način slijedi jedinstveni objektni model [1]. Ovakva vrsta aplikacije ne ovisi o nativnim komponentama uređaja, već dinamički stvara vlastite komponente. Pomoću vlastitog mehanizma za renderiranje pretvara kod aplikacije u nativni kod.



Slika 2 Struktura kontrola u Flutter-u [1]

2. Reaktivno programiranje

Reaktivno programiranje je paradigma programiranja koja se temelji na pojmu kontinuiranih, vremenski promjenjivih vrijednosti, tj. toka podataka i širenja promjena. Ova paradigma olakšava razvoj aplikacija temeljenih na događajima (engl. *Event-driven apps*) dopuštajući razvojnim programerima da izraze programe u terminima što treba činiti, a da jezik automatski upravlja tim kada će to učiniti [5]. Promjene stanja kod ove vrste paradigme se automatski i učinkovito šire kroz mrežu ovisnih izračuna, izračunatih od strane temeljnog modela izvršavanja. Drugim riječima, reaktivno programiranje je način kodiranja s asinkronim tokovima podataka koji olakšava kodiranje aplikacija i sučelja, koja dinamički reagiraju na promjene u podacima [6].

2.1. Koncepti

Biblioteka za asinkrono programiranje i programiranje temeljeno na događajima korištenjem nizova tokova podataka naziva se ReactiveX ili Reactive Extensions (skraćeno Rx). U Rx-u su implementirana načela reaktivnog programiranja. Asinkrono programiranje označava izvršavanje više blokova istovremeno, svaki blok koda se izvodi na vlastitoj niti. Programiranje temeljeno na događajima (engl. *event based*) označava izvršavanje koda na temelju generiranih događaja dok je program pokrenut, na primjer događaj klika elementa. Jednostavnim riječima, u Rx programiranju tokovi podataka, koje emitira jedna komponenta i temeljna struktura pružena od strane Rx-a, će propagirati te promjene na drugu komponentu koja je registrirana za primanje tih promjena podataka [7]. Rx se sastoji od tri ključne točke, dobavljača (engl. *Observable*), promatrača (engl. *Observer*) i raspoređivača (engl. *Scheduler*).

2.1.1. Dobavljač

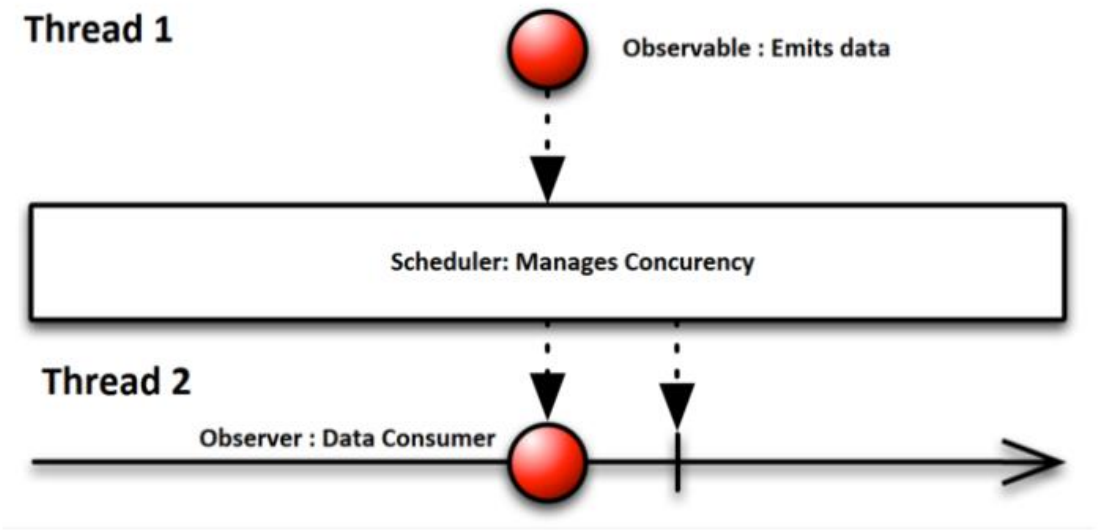
Dobavljač je zapravo izvor toka podataka. On pakira podatke koji se mogu prenositi iz jedne niti u drugu. U osnovi, emitira podatke povremeno ili samo jednom u svom životnom ciklusu na temelju svojih konfiguracija. Postoje različiti operatori koji dobavljaču mogu pomoći pri emitiranju specifičnih podataka na temelju određenih događaja. Ukratko, dobavljač obrađuje i dostavlja podatke drugim komponentama.

2.1.2. Promatrač

Promatrač je komponenta koja troši tok podataka koji emitira dobavljač. Promatrači se „pretplaćuju“ na dobavljača koristeći subscribeOn() metodu za primanje podataka koje dobavljač emitira. Često se definira kao pretplatnik (engl. *subscriber*). Prilikom emitiranja podataka, svi registrirani promatrači primaju podatke u onNext() povratnom pozivu (engl. *callback*) gdje mogu izvoditi razne operacije kao što je analiziranje JSON odgovora ili ažuriranje korisničkog sučelja. Prilikom pogreške kod emitiranja podataka od strane dobavljača, promatrač prima tu istu grešku u onError() metodu.

2.1.3. Raspoređivač

Pošto je Rx za asinkrono programiranje, potrebna je i komponenta za upravljanje dijelova procesa, odnosno niti. Treća ključna točka, odnosno komponenta zadužena za raspored izvršavanja je raspoređivač. On „govori“ dobavljačima i promatračima na kojoj niti se trebaju pokrenuti. Metodom observeOn() raspoređivač šalje informaciju promatraču koju nit treba promatrati, a metodom scheduleOn() šalje informaciju dobavljaču na kojoj se niti treba pokrenuti.



Slika 3 Tok podataka u Rx-u [7]

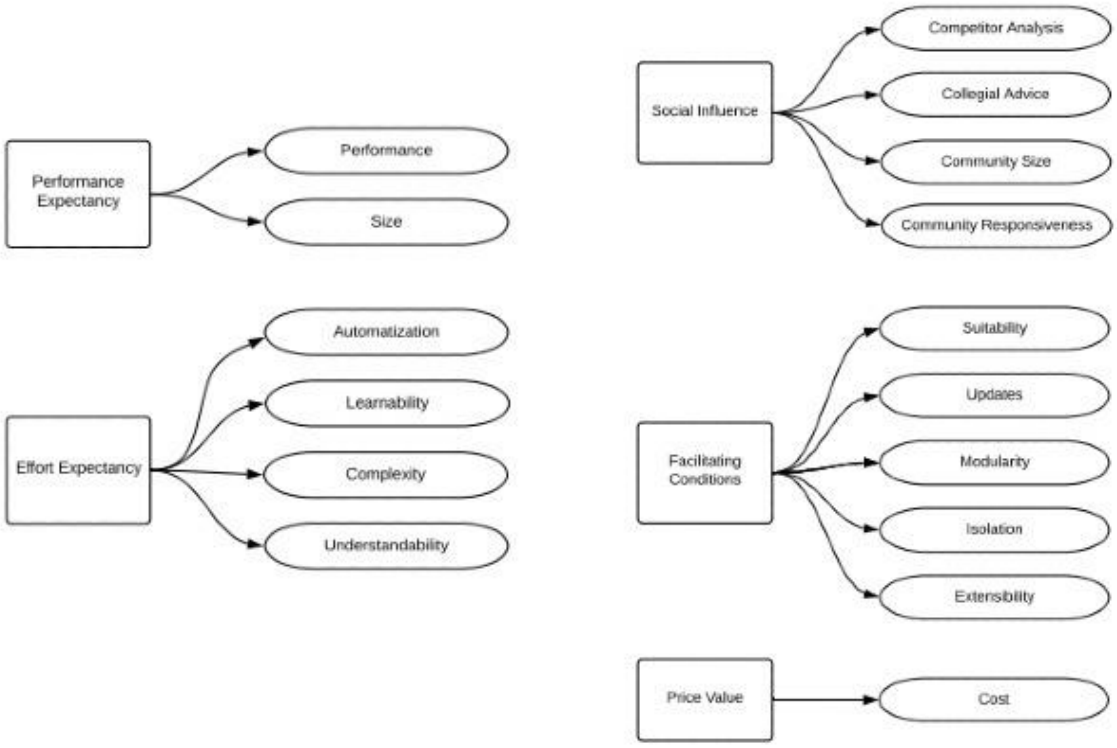
2.2. Okviri

U programiranju, okvir (engl. *framework*) predstavlja kostur ili nacrt koda koji pruža strukturu našoj aplikaciji. Može se smatrati predloškom koji se koristi pri početku rada na nekom projektu. Dva tipa okvira koja se koriste za izradu mrežnih i mobilnih aplikacija su pozadinski okvir (engl. *Backend*) i vanjski okvir (engl. *Frontend*). Backend je zapravo poslužiteljski dio aplikacije (engl. *server-side part*) u kojem se nalazi sva skrivena logika pokretanja aplikacije koja je u interakciji s korisnikom. Frontend je klijentski dio aplikacije (engl. *client-side part*) koji predstavlja interaktivno sučelje vidljivo korisniku. Postoji mnoštvo modernih okvira koji implementiraju model deklaracijskog renderiranja. Trenutno najpopularniji frontend okviri za izradu aplikacija su Angular, React, Vue i Svelte. Svi navedeni okviri slijede neku varijantu uzorka Model-Pogled-PogledModel (engl. *Model-View-ViewModel*, skraćeno MVVM) o kojem ćemo detaljnije u jednom od narednih potpoglavlja.

2.2.1. Faktori prihvaćanja

Početnici i iskusni programeri se suočavaju s nekoliko izazova kako bi identificirali elemente koje bi trebali uzeti u obzir pri odabiru okvira. Također, programeri koji žele izdati nove okvire zahtijevaju pregled najrelevantnijih karakteristika koje okvir mora osigurati kako bi se zajamčilo njegovo usvajanje. U svrhu lakšeg suočavanja s izazovima prilikom izbora okvira, provedeno je istraživanje tijekom kojeg je intervjuirano 18 sudionika [8]. Sudionici su bili programeri, vlasnici kompanija i poduzetnici.

Cilj samog istraživanja je bio doći do odgovora na pitanje „Koji čimbenici i akteri utječu na usvajanje JavaScript okvira?“. Analizom rezultata se došlo do traženog modela poželjnih čimbenika koji utječe na usvajanje okvira. Kategorije modela poželjnih faktora su: očekivani učinak, očekivani napor, društveni utjecaj, olakšavajući uvjeti i cijena. Također, rezultati su pokazali da kombinacija četiri aktera, u koje spadaju korisnik, programer, tim i vođa tima, utječu na usvajanje okvira.



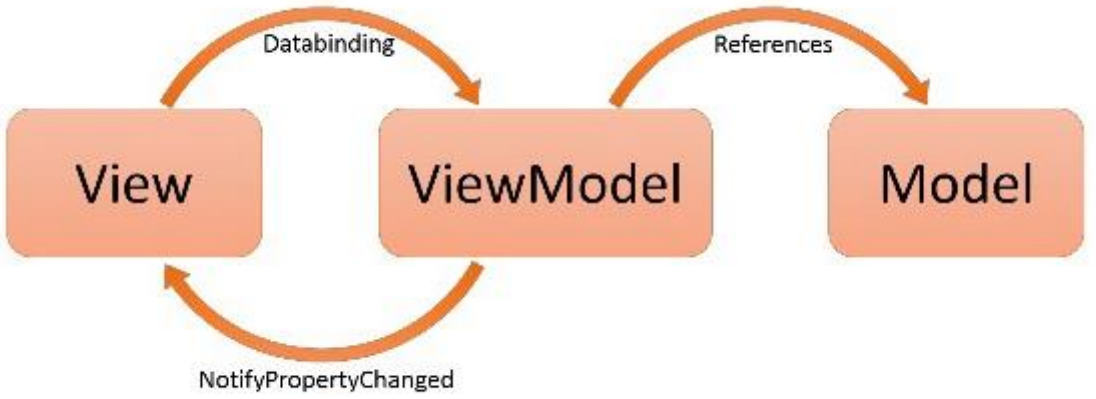
Slika 4 Model faktora koji utječe na usvajanje JavaScript okvira [8]

2.2.2. MVVM i povezivanje podataka

Model-Pogled-PogledModel (engl. *Model-View-ViewModel*, skraćeno MVVM) je deklarativni uzorak, gdje Model predstavlja izvor podataka aplikacije, a Pogled konkretno grafičko korisničko sučelje (engl. *Graphical User Interface*, skraćeno GUI) vidljivo korisniku. Pogled je izgrađen od PogledModela, koji sadrži deklarativni opis Pogleda zajedno sa svim relativnim podacima i logikom. Središnji koncept u MVVM-u je povezivanje podataka (engl. *data binding*), odnosno proces koji uspostavlja vezu između korisničkog sučelja aplikacije i podataka koje prikazuje [9]. Podaci unutar PogledModela se mogu povezati na Pogled tako da se sve njihove promjene automatski održavaju. U mrežnim okvirima (engl. *Web frameworks*) obično se koristi riječ komponenta (engl. *component*) umjesto PogledModel, ali su zapravo konceptualno ekvivalentni. U prethodno navedenim okvirima za izradu aplikaciju, svaka aplikacija je strukturirana kao stablo komponenti gdje svaka komponenta opisuje podskup DOM-a (engl. *Document Object Model*, skraćeno DOM).

DOM je platforma i jezično neutralno sučelje koje programima i skriptama omogućuje dinamički pristup i ažuriranje sadržaja, strukture i stila dokumenta [10]. Komponenta može sadržavati stanje aplikacije, koje po želji može dijeliti s komponentama potomcima (engl.

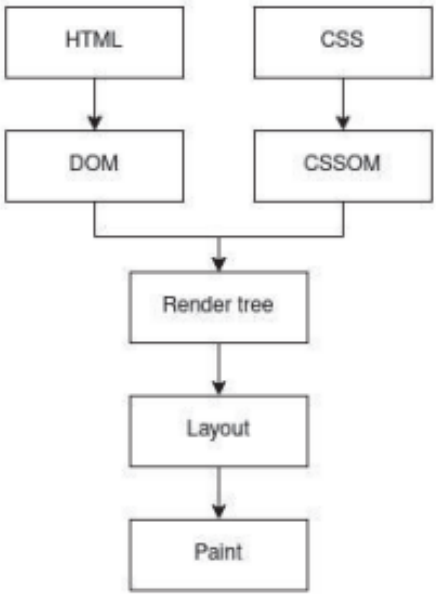
Descendant components), odnosno komponentama koje se, gledajući po strukturi stabla, nalaze na čvorovima ispod nje. Povezivanje podataka se koristi za definiranje prilagođene logike renderiranja. Na primjer, kod uvjetnog renderiranja (engl. *Conditional rendering*), prikaz nekog određenog elementa aplikacije može ovisiti o vrijednosti istinitosti (Boolean value) vezanog podatka. Vezanje podataka u navedenim okvirima uvijek je jednosmjerno, tok podataka teče od stanja aplikacije do korisničkog sučelja, obrnuto ne. To zapravo osigurava stanje cijele aplikacije, odnosno DOM manipulacija tijekom procesa renderiranja ne uzrokuje nikakvu promjenu stanja.



Slika 5 Prikaz interakcije u MVVM arhitekturi [12]

2.2.3. Usporedba renderiranja

Prilikom odabira okvira za izgradnju aplikacije, ključno je razumjeti temeljne karakteristike njezine strategije renderiranja. Općenito, u pregledu više okvira, postoje razlike u načinima ažuriranja postojećeg sadržaja. U provedenom istraživanju trenutno najpopularnijih okvira, prethodno navedenih u početku poglavlja 2.2, opisane su strategije renderiranja koje se koriste u tim okvirima [9]. Kako bi se spoznale razlike performansi, testirane su izvedbe okvira kritičnim putem renderiranja koji se koristi za renderiranje web stranica u pregledniku.



Slika 6 Kritični put renderiranja korišten za renderiranje web stranica u pregledniku [9]

Svi korišteni okviri su obavljali isti zadatak, koji je bio održavanje stanja DOM-a sinkroniziranog sa stanjem stabla komponenti stvaranjem i ažuriranjem DOM čvorova (engl. *nodes*) i njihovih atributa. Postoje dva načina koja opisuju kako se to postiže.

Prvi način, korišten u React-u i Vue-u, temelji se na eksplicitnom rješavanju problema udaljenosti u uređivanju stabla. Takav pristup se često naziva strategija renderiranja temeljena na virtualnom DOM-u (skraćeno vDOM), jer okviri upravljaju podatkovnom strukturom koja predstavlja određeno stanje DOM-a. U okviru koji se temelji na vDOM-u, svaka komponenta, kao svoj izlaz (engl. *output*), stvara jedan vDOM čvor koji opisuje skup DOM čvorova. U bilo kojem trenutku, moguće je proći stablom komponenti kako bi se produciralo novo vDOM stablo, koje zapravo predstavlja željeno stanje DOM-a na temelju trenutnog stanja aplikacije. Zatim se to stablo uspoređuje s prethodno generiranim vDOM stablom koje predstavlja trenutno, neažurirano stanje DOM-a. Nastala usporedba proizvodi skup promjena koje se moraju primijeniti na prethodno stablo da bi se dobilo novo. U konačnici, skup promjena se primjenjuje na pravi DOM pomoću DOM API-ja.

Drugi način , korišten kod Angular-a i Svelte-a, problem udaljenosti uređivanja stabla rješava implicitno. Kao i kod vDOM-a, svaka komponenta definira skup DOM čvorova koje treba prikazati instanca komponente. Za razliku od vDOM-a, niti u jednom trenutku ne postoji zaseban korak u kojem se izračunavaju sveukupne potrebne promjene korisničkog sučelja. Umjesto toga, svaka komponenta izravno modificira dio DOM-a koji predstavlja. Ovo se temelji na „prljavoj“ provjeri povezivanja podataka te dodatno, svaka

komponenta, osim praćenja trenutnog stanja aplikacije, također prati i vrijednosti svih procesa povezivanja podataka, kakve su trenutno predstavljene u DOM-u. Renderiranje se sastoji od prolaženja kroz stablo komponenti, izvođenja „prljavih“ provjera povezivanja podataka kako bi se vidjelo koja su se povezivanja promijenila i primjenjivanja promjena na DOM za svako pronađeno „prljavo“ povezivanje. Konačno rješenje problema udaljenosti uređivanja stabla je ukupni zbroj promjena.

Svi navedeni okviri izvode DOM ažuriranje u petlji renderiranja koja prolazi kroz stablo komponenti. Troškovi izvedbe petlje renderiranja ovise o veličinama ulaza i fiksnim troškovima. Veličine ulaza mogu se precizno mjeriti brojem komponenti koje svaka petlja mora obraditi, kao i brojem elemenata ili povezivanja podataka obrađenih po komponenti. Fiksni troškovi predstavljaju količinu posla koji je potrebno obaviti za svaki obrađeni element ili vezanje podataka. Petlja renderiranja može izvoditi dva različita posla, a to su stvaranje novih komponenti ili ažuriranje postojećih komponenti. Budući da komponente mogu dijeliti svoje stanje s komponentama potomcima, promjena stanja aplikacije u određenoj komponenti može utjecati i na izlaz drugih komponenti. Stoga, okviri moraju usvojiti strategiju kojom bi osigurali da su sve komponente procesuirane, odnosno obrađene prilikom promjene stanja aplikacije. Postoje tri pristupa pri rješavanju problema usvajanja te strategije.

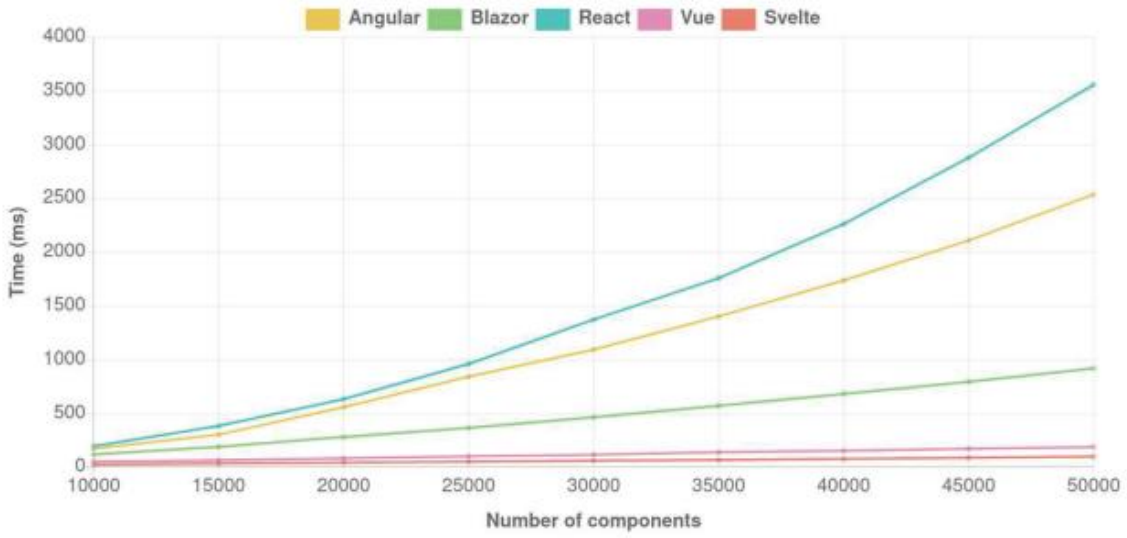
Prvi pristup, koji koristi Angular, je jednostavno proći kroz cijelo stablo komponenti točno jednom. Time se osigurava provjera svih veza podataka i obrada svih potrebnih promjena, ali zahtijeva nepotreban rad za bilo koje ažuriranje koje cilja samo na podskup stabla komponenti. U Angular-u je to moguće optimizirati ručnim označavanjem, tj. označiti kada se komponente ne bi smjele ponovno renderirati.

Drugi pristup, kojeg koristi React, je prolazak kroz podstablo komponente koja pokreće petlju renderiranja. Komponentama je dopušteno dijeliti svoje stanje samo sa svojim potomcima. To znači da promjena stanja u nekoj komponenti ne može utjecati na komponente koje se nalaze iznad stabla te komponente. Iz tog razloga je zajamčeno da će ovaj pristup obraditi sve potrebne komponente, ali će i dalje zahtijevati nepotreban rad za potomke čiji se izlaz nije promijenio. Kao i kod Angular-a, i React omogućuje programeru aplikacije da ručno odredi kada se ponovno renderiranje komponente može sigurno priskočiti.

Konačni, treći pristup, koji su poduzeli Vue i Svelte, je obrada samo „prljavih“ komponenti, odnosno komponenti čiji se izlaz promijenio. Ovaj pristup je optimalan, ali na

neki način zahtijeva određenost „prljavih“ komponenti unaprijed. I Vue i Svelte to postižu korištenjem ograničenog sustava reaktivnosti. Izlaz komponente tretira se kao dio grafa ovisnosti, gdje se svaka vrijednost koja utječe na izlaz tretira kao ovisnost o komponenti. Kad god se takva vrijednost promijeni ili ponovno dodijeli, sustav reaktivnosti automatski označava komponentu kao prljavu i zakazuje njezino ponovno renderiranje. Ovo radi tranzitivno među komponentama koje ovise o ulazima dodijeljenih od strane nadređene komponente (engl. *parent component*), osiguravajući da se sve zahvaćene komponente automatski označavaju prljavima te nema potrebe za provjerom nezahvaćenih komponenti. Vue-ov sustav reaktivnosti temelji se na eksplicitnom praćenju ovisnosti tijekom izvođenja pomoću proxyja, posebne vrste JavaScript objekta koji omogućuje pristup presretanja drugim objektima tijekom izvođenja. Svelte-ov sustav reaktivnosti, nasuprot tome, prati grafikone ovisnosti samo u vrijeme kompiliranja i radi generiranjem imperativnog koda koji tijekom vremena izvođenja ažurira grafikone ovisnosti kad god se vrijednosti ponovno dodjeljuju ili mijenjaju. Funkcionalno, oba su pristupa ekvivalentna, ali Svelte-ov pristup ima potencijalno manje troškove rada.

Izlaz komponente može se podijeliti na statički i dinamički sadržaj, gdje se statički sadržaj nikada neće mijenjati nakon početnog renderiranja komponente, dok se dinamički sadržaj, sadržaj koji ovisi o povezivanju podataka, može mijenjati. Od navedenih okvira, samo React obrađuje i statički i dinamički sadržaj prilikom svakog renderiranja, dok svi ostali okviri obrađuju statički sadržaj samo prilikom početnog renderiranja komponente.



Slika 7 Vrijeme izvršavanja skripte u milisekundama pri unosu N elemenata u jednu komponentu [9]

Analizom rezultata izvedbi utvrđeno je da će okviri, koji obrađuju statički sadržaj u svakoj petlji renderiranja, imati vjerojatno najlošiju izvedbu. Okvir Svelte je kod svakog mjerenja imao najbolje performanse od svih testiranih okvira. Identificirani faktori koji utječu na poboljšanu izvedbu okvira su:

- Upotreba reaktivnog sustava za automatsko otkrivanje „prljavih“ komponenti
- Upotreba optimalnog kompilatora za generiranje ažurnog koda komponente koji zanemaruje statički sadržaj
- Upotreba renderiranja temeljenog na povezivanju podataka, umjesto virtualnog DOM-a

3. Implementacija mobilne aplikacije u okvirima React Native i Svelte Native

U praktičnom dijelu rada, implementirane su dvije mobilne aplikacije na temu „Skladište“. Za izradu prve aplikacije korišten je okvir React Native, a za izradu druge aplikacije korišten je okvir Svelte Native.

3.1. Opis aplikacije

Aplikacija korisniku služi za vođenje evidencije o isporukama u skladištu. Korisnik se može prijaviti u sustav putem unosa podataka postojećeg korisnika. Postoje tri razine prijave u sustav, odnosno postoje tri tipa korisnika:

- Admin,
- Radnik
- Poslovođa

Admin ima pregled u korisničke račune, te ima mogućnost modificiranja korisničkih računa. Radnik ima uvid u neisporučene isporuke te ih može evidentirati ukoliko su pristigle u skladište, dok poslovođa ima mogućnost brisanja ili uređivanja starih i dodavanja nadolazećih isporuka.

3.2. React Native aplikacija

Prvi korišteni okvir, React Native, je zapravo okvir koji koristi React.js biblioteku za izradu mobilnih korisničkih sučelja. Od Reacta se razlikuje po tome što, kao gradivne blokove, koristi nativne komponente umjesto web komponenti.

3.2.1. Struktura i povezivanje podataka

U glavnoj komponenti aplikacije App(), koja se nalazi unutar datoteke App.js u kojoj se odvija cijela logika aplikacije, ugniježđena je komponenta Provider uvedena iz React Redux-a, bibiloteke koja React komponentama omogućava čitanje i slanje radnji za

ažuriranje podataka iz spremnika, tj. cjelokupnog stanja aplikacije. Dakle, komponenta Provider čini spremnik dostupan svim ugniježđenim komponentama koje trebaju pristup spremniku. Komponenta koja se nalazi unutar Providera, odnosno na drugoj najvišoj razini stabla glavne komponente, je NavigationContainer koji je odgovoran za upravljanje stanjem aplikacije, tj. zaslužan je za povezivanje navigatora najviše razine s Providerom. Unutar NavigationContainera ugniježđen je navigator koji aplikaciji omogućuje prijelaz između zaslona. Na zadnjoj razini nalaze se ekrani umetnuti unutar navigator komponente. Svaki ekran (engl. screen) sadrži svojstva ime (engl. *name*) i komponentu (engl. *component*). svojstvo ime prima naziv rute ekrana koji služi za samu logiku navigacije među ekranima, a svojstvo komponenta prima ime komponente za prikaz koja je uvezena u glavni dio aplikacije.

```
export default function App() {
  return (
    <Provider store={centralniSpremnik}>
      <NavigationContainer>
        <Stack.Navigator>
          <Stack.Screen name="Skladiste" component={PocetnaEkran} />
          <Stack.Screen name="Login" component={LoginEkran} />
          <Stack.Screen name="admin" component={PocetnaAdminEkran} />
          <Stack.Screen name="poslovoda" component={PocetnaPoslovodaEkran} />
          <Stack.Screen name="radnik" component={PocetnaRadnikEkran} />
          <Stack.Screen name="UnosIsporuke" component={UnosIsporukeEkran} />
          <Stack.Screen name="UnosKorisnika" component={UnosKorisnikaEkran} />
          <Stack.Screen name="PrikazIsporuka" component={PrikazIsporuka} />
          <Stack.Screen name="PrikazKorisnika" component={PrikazKorisnika} />
        </Stack.Navigator>
      </NavigationContainer>
    </Provider>
  );
}
```

Slika 8 Struktura glavne komponente aplikacije

Svaka komponenta je zapravo JavaScript XML (skraćeno JSX) element. JSX je proširenje sintakse za JavaScript koji kreira React elemente. On omogućuje pisanje HTML-a u React-u, tj. pretvara HTML oznake u React elemente.

```
const LoginEkran = ({navigation}) => {

    const [username, postaviUsername] = useState('')
    const [pass, postaviPass] = useState('')

    const korisnici = useSelector( (state) => state.korisnici.korisnici)
    const logiraniKorisnik = useSelector ( (state) => state.login.korisnik)

    const dispatch = useDispatch()
    const promjenaUsername = (e) => {
        postaviUsername(e)
    }
    const promjenaPass = (e) => {
        postaviPass(e)
    }

    const prijava = () => {
        let prijavljen = false

        korisnici.map(k => {
            if (k.username === username && k.pass === pass) {
                dispatch(prijavaKorisnika(k))
                navigation.navigate(`${k.uloga}`, {
                    ime: k.ime,
                    prezime: k.prezime
                });
                prijavljen = true
            }
        })
        if(prijavljen === false) {
            alert('Neispravni podaci')
        }

    }

    return (
        <View style={stil.ekran}>
            <View>
                <Text>Korisnicko ime: </Text>
                <TextInput
                    style={stil.unos}
                    value={username}
                    onChangeText={promjenaUsername}
                />
            </View>
            <View>
                <Text>Lozinka: </Text>
                <TextInput
                    style={stil.unos}
                    secureTextEntry={true}
                    value={pass}
                    onChangeText={promjenaPass}
                />
            </View>
            <Tipka
                onPress={() => prijava()}
                title={'Prijavi se'}
                style={stil.tipka}
            />
        </View>
    )
}
```

Slika 9 Primjer React komponente u JS datoteci

Iz priložene slike 3.2 vidimo da se pri vrhu funkcijske komponente nalaze sva početna stanja komponente implementirana React useState hook-om. UseState hook je funkcija

koja omogućava praćenje stanja u funkcijskoj komponenti. Stanje inicijaliziramo tako što prvo definiramo varijablu koja označava trenutno stanje te naziv funkcije koja ažurira trenutno stanje, a zatim pozivamo useState hook kojem dodjeljujemo vrijednost početnog stanja. Aktiviranjem događaja koji poziva funkciju ažuriranja stanja, mijenja se određeno stanje aplikacije. To zapravo predstavlja logiku povezivanja podataka u React komponenti.

U ovom primjeru, izraz kojeg ova komponenta vraća je root komponenta View koja je zapravo React Native (skraćeno RN) komponenta. Unutar nje su ugniježđene i ostale RN komponente uvezene iz RN biblioteka. Uvoz potrebnih elemenata se inicijalizira na početku datoteke u kojoj se komponenta nalazi.

```
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux'
import { prijavaKorisnika } from '../store/actions/login';
import { View, Text, StyleSheet, Dimensions, TextInput } from 'react-native';
import Tipka from '../components/Tipka';
```

Slika 10 Primjer uvoza komponenti i funkcija u JS datoteci

Sve RN komponente se mogu oblikovati pomoću StyleSheet objekta koji je zapravo apstrakcija slična CSS StyleSheetu. Metodom create() stvaramo željeni stil kojeg imenujemo i referenciramo pod svojstvo style željene RN komponente. Stilski objekt se instancira izvan funkcionalne komponente, na kraju datoteke.

```
const stil = StyleSheet.create({
    ekran: {
        flex: 1,
        backgroundColor: '#F0F8FF',
        justifyContent: 'center',
        padding: Dimensions.get('window').width / 5
    },
    unos: {
        fontSize: 16,
        borderColor: 'black',
        borderWidth: 1,
        borderRadius: 6,
        paddingTop: '2%',
        paddingBottom: '2%',
        marginTop: 10,
        marginBottom: 20
    },
    tipka: {
        backgroundColor: 'lightgreen',
        marginTop: 20,
    }
})
```

Slika 11 Primjer StyleSheet objekta u JS datoteci

3.2.2. Navigacija i upravljanje stanjem

Navigacijska logika je implementirana u svim React komponentama koje su uvezene pod svojstvo „component“ unutar komponente Stack.Screen, koja je ugniježđena u komponentu Stack.Navigator u glavnom dijelu aplikacije. Navedene komponente sadrže svojstvo navigation kojeg prosljeđuju Stack.Screen elementu. Svojstvo navigation inicijaliziramo unutar props objekta komponente. Props objekt možemo definirati kao argument za prosljeđivanje. Svojstvom navigation pozivamo metodu navigate() koja kao parametar prima ime rute ekrana na kojeg želimo prijeći. Također unutar metode, osim imena rute Stack.Screen komponente, možemo slati i podatke. Podatke šaljemo u obliku objekta s parametrima. Svaki ekran na koji smo se prebacili ima dostupno svojstvo route s atributom params u kojem se nalaze parametri poslani iz prethodnog ekrana. Svojstvo route, kao i navigation, inicijaliziramo unutar props objekta komponente koja prima podatke.

```
const PocetnaAdminEkran = ({ route, navigation }) => {

    const {ime, prezime} = route.params

    const dispatch = useDispatch()

    const odjava = () => {
        dispatch(odjavaKorisnika())
        navigation.navigate('Skladiste')
    }

    return (
        <View style={stil.glavniEkran}>
            <View style={stil.ekran}>
                <View>
                    <Text style={stil.text}>Prijavljeni ste kao:</Text>
                    <Text style={stil.username}>{ime} {prezime}</Text>
                </View>
                <Tipka
                    style={stil.tipka}
                    title="Odjavi se"
                    onPress={() => odjava()}
                />
            </View>
            <Tipka
                onPress={() => {
                    navigation.navigate('UnosKorisnika')
                }}
                title={'UNOS KORISNIKA'}
                style={stil.korisniciTipka}
                stilNaslov={stil.stilNaslov}
            />
            <Tipka
                onPress={() => {
                    navigation.navigate('PrikazKorisnika')
                }}
                title={'KORISNICI'}
                style={stil.korisniciTipka}
                stilNaslov={stil.stilNaslov}
            />
        </View>
    );
}
```

Slika 12 Primjer komponente ekrana sa svojstvima route i navigation

Za implementaciju spremnika korištena je biblioteka Redux koja služi za upravljanje stanjem aplikacije. Stanje aplikacije pohranjeno je u spremniku (engl. *store*). Za osvježavanje stanja aplikacije, spremnik koristi reducer-e. Reducer je funkcija koja određuje promjene stanja aplikacije. Svaki reducer koristi radnju ili akciju (engl. *action*) kojom, ovisno o tipu, mijenja trenutno stanje spremnika. Pozivom akcije unutar komponente poziva se određena reducer funkcija koja ažurira stanje spremnika. Prilikom ažuriranja, spremnik šalje obavijest o promjeni stanja svim komponentama promatračima nakon čega se osvježavaju podaci na sučelju.

```
const glavniReducer = combineReducers({
  isporuke: isporukaReducer,
  korisnici: korisnikReducer,
  login: loginReducer
})

const centralniSpremnik = createStore(glavniReducer)


export default function App() {
  return (
    <Provider store={centralniSpremnik}>
```

Slika 13 Primjer implementacije spremnika u React Native-u

```
import * as React from 'react';
import { useDispatch } from 'react-redux'
import { odjavaKorisnika } from '../store/actions/login';
import { View, Text, StyleSheet, Dimensions } from 'react-native';
import Tipka from '../components/Tipka';

const PocetnaAdminEkran = ({ route, navigation }) => {

    const {ime, prezime} = route.params

    const dispatch = useDispatch()

    const odjava = () => {
        dispatch(odjavaKorisnika())
        navigation.navigate('Skladiste')
    }

    return (
        <View style={stil.glavniEkran}>
            <View style={stil.ekran}>
                <View>
                    <Text style={stil.text}>Prijavljeni ste kao:</Text>
                    <Text style={stil.username}>{ime} {prezime}</Text>
                </View>
                <Tipka
                    style={stil.tipka}
                    title="Odjavi se"
                    onPress={() => odjava()}
                />
```

Slika 14 Primjer poziva akcije u React Native-u

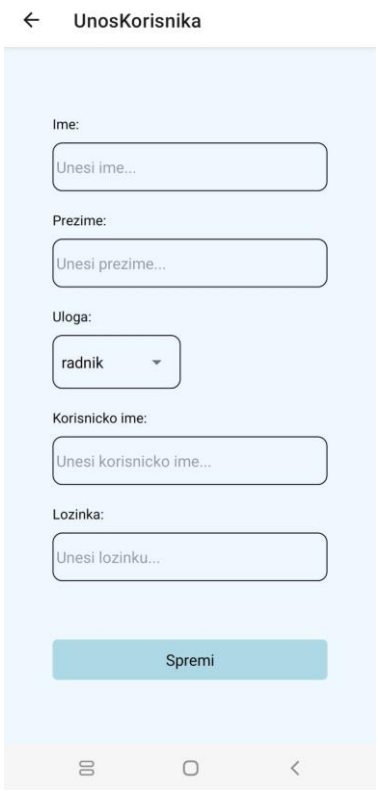
Iz primjera uočavamo korištenje dispatch metode unutar koje pozivamo uvezenu akciju. Funkcijom dispatch se zapravo pokreće ažuriranje spremnika.

Izgled React Native aplikacije „Skladište“ prikazan je na sljedećim slikama.

Slika 15 Početni ekran RN

Slika 16 Ekran za prijavu RN

Slika 17 Početni ekran admina RN



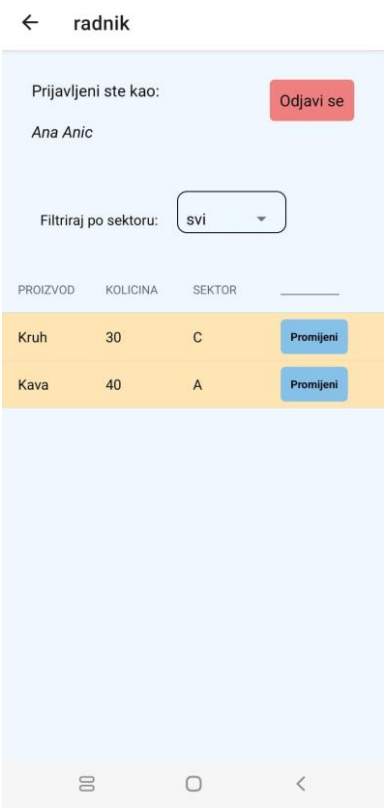
Slika 18 Ekran za registraciju korisnika RN

Slika 19 Ekran za prikaz korisnika RN

Slika 20 Početni ekran poslovođe RN

Slika 21 Ekran za unos isporuke RN

Slika 22 Ekran za prikaz isporuka RN



Slika 23 Početni ekran radnika RN

3.3. Svelte Native aplikacija

Pri izradi druge aplikacije korišten je okvir Svelte Native. Svelte Native je zapravo novi pristup koji se koristi za razvoj nativnih IOS I Android aplikacija. Nastao je spajanjem tehnologija Svelte i NativeScript.

3.3.1. Struktura i povezivanje podataka

TypeScript datoteka app.ts je ulazna točka aplikacije. TypeScript je programski jezik koji se može definirati kao JavaScript s dodanim značajkama. Prevodi se u JavaScript. Primarna svrha navedene datoteke je prijenos kontrole na prvu stranicu aplikacije.

app > TS app.ts

```
1  import { svelteNativeNoFrame } from 'svelte-native'
2  import App from './App.svelte'
3
4  svelteNativeNoFrame(App, {})
5
```

Slika 24 Prikaz datoteke app.ts

Iz priložene slike vidimo da glavna funkcija za pokretanje aplikacije svelteNativeNoFrame() kao prvi parametar prima komponentu za renderiranje, odnosno Svelte komponentu App.

app > App.svelte > ...

```
1  <script lang="ts">
2    import PocetnaPage from "./pages/PocetnaPage.svelte";
3  </script>
4
5  <frame>
6    <PocetnaPage />
7  </frame>
8
```

Slika 25 Prikaz datoteke App.svelte

Pregledom App.svelte datoteke vidimo da se pri početku nalazi script oznaka unutar koje pišemo sav kod koji se odnosi na komponentu van skripte. Ovdje je upotrijebljena za uvoz svelte datoteke. Ova Svelte komponenta kao izraz vraća okvir (engl. *frame*) komponentu u kojoj se odvija cijela navigacija aplikacije. U ovom slučaju, root element okvira predstavlja Svelte komponenta PocetnaPage koja označava početnu stranicu (engl. *default page*) aplikacije.

Struktura svelte datoteke koja definira stranicu aplikacije se sastoji od već navedene HTML script oznake u kojoj instanciramo sva stanja, funkcije i varijable koje upućuju na sadržaj izvan skripte, page komponente koja čini sadržaj i HTML style oznake unutar koje oblikujemo komponente sadržaja.

```
<page>
    <actionBar class="barNaslovna" title="Pretraga Isporuka" />
    <gridLayout>
        <stackLayout>
            <button class="returnBtn" text="⇦" on:tap={goBack}/>
            <stackLayout class="pretragaIsporukeStack">
                <stackLayout class="searchStack">
                    <searchBar hint="Pretraži po proizvodu..." bind:text={proizvod} />
                </stackLayout>
                <stackLayout class="filterStack" orientation="horizontal">
                    <label class="labelFilter" text="Pretraži po sektoru:"/>
                    <listPicker
                        class="filterSektor"
                        items={listaSektora}
                        bind:selectedValue={sektor}

                    />
                </stackLayout>
                <stackLayout>
                    <button class="pretraziBtn" text="PRETRAŽI"
                        on:tap={ () => {
                            pretraziIsporuke()
                            navigate({ page: PrikazIsporukaPage})
                            }
                        }
                    />
                </stackLayout>
            </stackLayout>
        </stackLayout>
    </gridLayout>
</page>
```

Slika 26 Primjer sadržaja stranice u svelte datoteci

U Svelte Native-u, Svelte komponenta može predstavljati kombinaciju dvije vrste komponenata:

- NativeScript komponente
- Svelte Native komponente

Nazivi NativeScript komponenti počinju malim početnim slovom, dok nazivi Svelte Native komponenti počinju velikim početnim slovom. Razlog pisanja takve sintakse je davanje do znanja Svelte kompilatoru o kojim se komponentama radi. Iz priložene komponente uočavamo da je princip povezivanja podataka u Svelteu drugačiji od onog u Reactu. Svelte omogućuje povezivanje podataka pomoću bind funkcije. Funkcionira na način da na svaku promjenu vrijednosti proslijeđenog svojstva elementa u kojem je pozvana, ažurira vrijednost ciljane varijable, definirane u skripti, s vrijednošću svojstva elementa. U narednim slikama prikazana su i preostala dva dijela svelte datoteke.

```
<script>
    import spremnik from "~/store/centralniSpremnik";
    import { pretragaIsporuka } from "~/store/actions/isporuke";
    import { navigate, goBack } from "svelte-native";
    import PrikazIsporukaPage from "./PrikazIsporukaPage.svelte";

    let proizvod = ""
    let listaSektora = ['svi', 'A', 'B', 'C']
    let sektor = ''

    const pretraziIsporuke = () => {
        spremnik.dispatch(pretragaIsporuka(sektor, proizvod))
    }

</script>
```

Slika 27 Primjer script oznake unutar svelte datoteke

```
<style>
    .barNaslovna{
        background-color: #658ff1;
        color: white;
    }
    .returnBtn {
        margin-top: 15;
        margin-left: 10;
        padding-top: 6%;
        horizontal-align: left;
        border-radius: 10;
        background-color: #658ff1;
        color: whitesmoke;
        font-size: 20;
        width: 70;
        height: 110px;
    }
    .pretragaIsporukeStack {
        padding: 30%;
        margin-top: 50;
        horizontal-align: center
    }
    .searchStack {
        font-size: 16;
    }
    .filterStack {
        margin-top: 50;
        font-size: 16;
    }
    .labelFilter {
        margin-top: 105;
        margin-left: 10;
        margin-right: 40;
    }
    .filterSektor {
        height: 105;
        width: 120;
        border-color: black;
        border-width: 1;
        border-radius: 10;
        margin: 65 0
    }
    .pretraziBtn {
        margin-top: 70;
        height: 50;
        border-radius: 5;
        background-color: lightgray;
        color: white;
    }

</style>
```

Slika 28 Primjer sytle oznake unutar svelte datoteke

Također, Svelte kao i React, nudi mogućnost definiranja props objekta čiju vrijednost možemo definirati unutar neke druge komponente. Kako bi ta logika funkcionirala u Svelte Native-u, potreban je izvoz (engl. *export*) props objekta.

```
<script>
    export let isporuka;
    export let brisanje;
    export let promjenaStatusa;

</script>

<stackLayout orientation="horizontal" backgroundColor={isporuka.status ? "#65a9f1" : "#FFECC5"}>
    <label class="celija" text={isporuka.proizvod} width= "25%" />
    <label class="celija" text={isporuka.kolicina} width= "17.5%" />
    <label class="celija" text={isporuka.sektor} width= "17.5%" />
    <button on:tap={brisanje} class="btnBrisi" text="BRIŠI" width= "20%" />
    <button on:tap={promjenaStatusa}
            class={isporuka.status ? "btnPromijeniIsp" : "btnPromijeniNeisp"}
            text="PROMIJENI"
            width= "20%"
    />
</stackLayout>

<style>
    .celija {
        margin-top: 15;
        text-align: center;
    }
    .btnBrisi {
        font-size: 9.75;
        background-color: rgb(255, 183, 0)
    }
    .btnPromijeniIsp {
        font-size: 9.75;
        background-color: #FFECC5;
    }
    .btnPromijeniNeisp {
        font-size: 9.75;
        background-color: #65a9f1
    }
</style>
```

Slika 29 Primjer Svelte komponente s props objektom

3.3.2. Navigacija i upravljanje stanjem

Logika navigacije među komponentama u Svelte Native-u je implementirana metodom navigate() koja pod svojstvo page prima ime stranice na koju želimo prijeći. Ime stranice označava uvezenu Svelte komponentu. Također, osim metode navigate, Svelte Native nudi i metodu goBack(). Pozivom nje zaslon se prebacuje na prethodno pregledanu stranicu.

```
<script>
    import spremnik from "~/store/centralniSpremnik";
    import { prijavaKorisnika } from "~/store/actions/login";
    import { navigate, goBack } from "svelte-native";
    import PocetnaPoslovodaPage from "./PocetnaPoslovodaPage.svelte";
    import PocetnaAdminPage from "./PocetnaAdminPage.svelte";
    import PocetnaRadnikPage from "./PocetnaRadnikPage.svelte";

    let stanje = spremnik.getState()
    let korisnici = stanje.korisnici.korisnici

    let username = "";
    let pass = "";

    const prijava = () => {
        let prijavljen = false
        korisnici.map(k => {
            if (k.username === username && k.pass === pass) {
                if (k.uloga === "poslovoda") {
                    spremnik.dispatch(prijavaKorisnika(k))
                    navigate({ page: PocetnaPoslovodaPage })
                    prijavljen = true
                }
                if (k.uloga === "admin") {
                    spremnik.dispatch(prijavaKorisnika(k))
                    navigate({ page: PocetnaAdminPage })
                    prijavljen = true
                }
                if (k.uloga === "radnik") {
                    spremnik.dispatch(prijavaKorisnika(k))
                    navigate({ page: PocetnaRadnikPage })
                    prijavljen = true
                }
            }
        })
        if(prijavljen === false) {
            alert('Neispravni podaci')
        }
    }
</script>

<page>
    <actionBar class="barLogin" title="Login" />
    <gridLayout class="glavniGrid">
        <stackLayout>
            <button class="returnBtn" text="" on:tap="{goBack}" />
        </stackLayout>
        <stackLayout>
            <stackLayout class="loginStack">
                <label text="Korisnicko ime:" />
                <stackLayout>
                    <textField class="okvir" bind:text={username}/>
                </stackLayout>
                <label text="Lozinka:" />
                <stackLayout>
                    <textField class="okvir" bind:text={pass} secure={true} />
                </stackLayout>
                <button
                    class="prijaviSeBtn"
                    text="PRIJAVI SE"
                    on:tap={() => prijava()}
                />
```

Slika 30 Primjer preusmjeravanja u Svelte komponenti

Za upravljanje stanjem, kao i kod Reacta, korištena je Redux biblioteka. Stanje aplikacije pohranjeno je u spremniku koji za ažuriranje svog stanja koristi reducer funkcije koje su već prethodno objašnjene kod upravljanja stanjem u React Native aplikaciji.

app > store > JS centralniSpremnik.js > default

```
1   import { createStore, combineReducers } from "redux";
2
3   import isporukaReducer from "./reducers/isporuke";
4   import korisnikReducer from "./reducers/korisnici";
5   import loginReducer from "./reducers/login";
6
7   const glavniReducer = combineReducers({
8       isporuke: isporukaReducer,
9       korisnici: korisnikReducer,
10      login: loginReducer
11  })
12
13  const centralniSpremnik = createStore(glavniReducer)
14
15  export default centralniSpremnik;
```

Slika 31 Primjer kreiranja spremnika

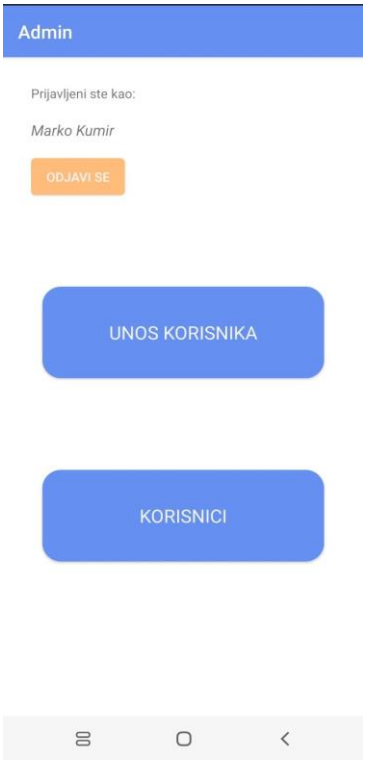
```
<script>
    import spremnik from '~/store/centralniSpremnik';
    import { odjavaKorisnika } from '~/store/actions/login';
    import { navigate } from 'svelte-native'
    import PocetnaPage from './PocetnaPage.svelte'
    import UnosKorisnika from './UnosKorisnikaPage.svelte'
    import PrikazKorisnikaPage from './PrikazKorisnikaPage.svelte';

    let stanje = spremnik.getState()
    let logiraniKorisnik = stanje.login.korisnik
    let ime = logiraniKorisnik.ime
    let prezime = logiraniKorisnik.prezime

    const odjava = () => {
        spremnik.dispatch(odjavaKorisnika())
        navigate({ page: PocetnaPage})
    }

</script>

<page>
    <actionBar class="barNaslovna" title="Admin" />
    <gridLayout class='glavniGrid'>
        <stackLayout>
            <label text='Prijavljeni ste kao:' />
            <label class="imePrezime" text="{ime} {prezime}"/>
            <button class='odjavaBtn' text="ODJAVI SE"
                on:tap={() => odjava()}
            />
```

Slika 32 Primjer poziva akcije u Svelte Native-u

Na sljedećim slikama prikazan je izgled Svelte Native aplikacije „Skladište“ iz kojih možemo vidjeti da se ne razlikuje previše u odnosu na React Native verziju.

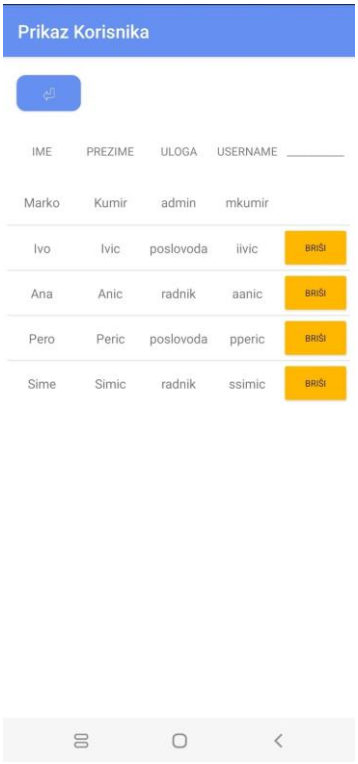
Slika 33 Početni ekran SN

Slika 34 Ekran za prijavu SN



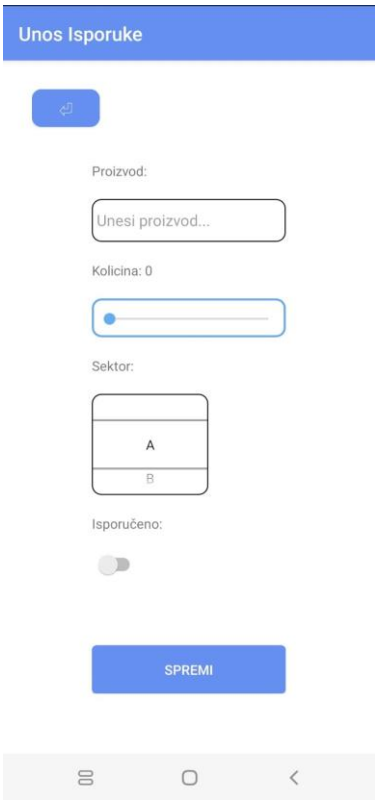
Slika 35 Početni ekran admina SN

Slika 36 Ekran za registraciju korisnika SN



Slika 37 Ekran za prikaz korisnika SN

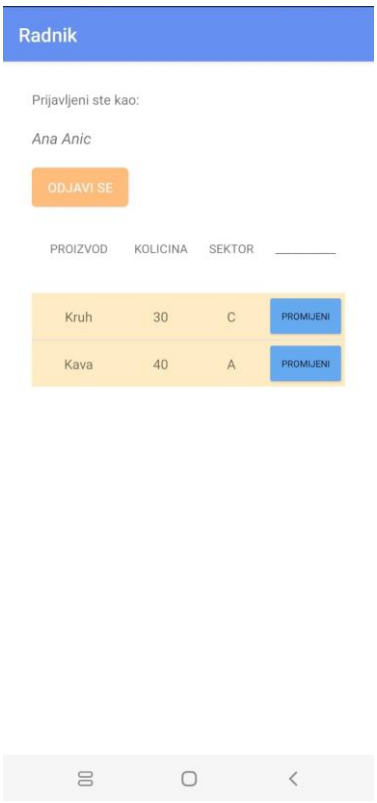
Slika 38 Početni ekran poslovođe SN



Slika 39 Ekran za unos isporuke SN

Slika 40 Ekran za pretragu isporuka SN

Slika 41 Ekran za prikaz isporuka SN



Slika 42 Početni ekran radnika SN

Ova aplikacija, za razliku od React Native-a, sadrži dodatni ekran za pretragu isporuka.

3.4. Usporedba

Kao što je već spomenuto, React Native i Svelte Native su okviri koji služe za razvoj Android i iOS aplikacija. U narednim tablicama prikazane su usporedbe navedena dva okvira i tehnologija koje ih pokreću.

	REACT NATIVE	**SVELTE NATIVE**
Vrste komponenti	RN komponente	NS i SN komponente
Oblikovanje	RN StyleSheet	CSS
Navigacija	Stack.Navigator	frame
Povezivanje podataka	useState	bind
Izvoz datoteke	Da	Ne
Ekstenzija datoteke	.js	.svelte
Alat za pregled aplikacije	Expo	NativeScript Preview
Godina kreiranja	2015.	2019.
Mjesečna paketna preuzimanja [11]	4.453.786	1.340

	REACT	**SVELTE**
Sintaksa	JSX	[HTML, CSS, JavaScript]
Strategija renderiranja	virtualni DOM	stvarni DOM
Izvedba	dobra	vrlo dobra
Istaknuta kvaliteta	robusnost	brzina
Reaktivnost	velika	jako velika
Godina kreiranja	2011.	2016.

Zaključak

Sve veća pojava pametnih telefona (engl. *smartphone*), zajedno s aplikacijama koje se na njima pokreću, možda je najveći tehnološki fenomen u posljednje vrijeme. Mobilne aplikacije su evoulirale iz običnog sredstva komunikacije u vrlo bitan poslovan alat. Čine korištenje tehnologije izvedivim i razumljivim. Korisnicima omogućuju jednostavan, funkcionalan pristup informacijama, uslugama i proizvodima koji su im potrebni.

Često, mobilne aplikacije su specifično kodirane za određeni mobilni uređaj, kako bi se iskoristile prednosti jedinstvenih značajki uređaja. Za ubrzanje samog procesa izrade, programeri koriste okvire koji služe kao temelj pri samom razvoju mobilnih aplikacija. Svojm skupom biblioteka pružaju jednostavniju i bržu izradu.

Ideja ovog rada je bila usporediti dva različita mobilna okvira, React Native i Svelte Native. Kroz zadnje poglavlje, uspoređena je njihova struktura komponenti i datoteka, logika povezivanja podataka, navigacija i upravljanje stanjem. U posljednjem potpoglavlju, prikazana je kratka usporedba navedenih okvira na temelju raznih parametara u obliku tablice.

Literatura

[1] K. Shah, H. Sinha, and P. Mishra, ‘Analysis of Cross-Platform Mobile App Development Tools’, in *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, Bombay, India, Mar. 2019, pp. 1–7. doi: 10.1109/I2CT45611.2019.9033872.

[2] Firtman, Maximiliano. High Performance Mobile Web: Best Practices for Optimizing Mobile Web Apps. " O'Reilly Media, Inc.", 2016.

[3] Sánchez Blanco, Andrea. "Development of hybrid mobile apps: Using Ionic Framework.", 2016.

[4] Xanthopoulos, Spyros, and Stelios Xinogalos. "A comparative analysis of cross-platform development approaches for mobile applications." Proceedings of the 6th Balkan Conference in Informatics. 2013.

[5] E. Bainomugisha, A. L. Carreton, T. van Cutsem, S. Mostinckx, and W. de Meuter, ‘A survey on reactive programming’, *ACM Comput. Surv.*, vol. 45, no. 4, pp. 1–34, Aug. 2013, doi: 10.1145/2501654.2501666.

[6] Medium, Mahdi Chtioui: „ReactiveX: Reactive Programming Principles“

https://medium.com/@mahdichtioui/reactivex-reactive-programming-principles-dbb1bafa8384

[7] Medium, Keval Patel: „What is Reactive Programming?“

What is Reactive Programming?. Nowadays everybody is talking about… | by Keval Patel | Medium

[8] A. Pano, D. Graziotin, and P. Abrahamsson, ‘Factors and actors leading to the adoption of a JavaScript framework’, *Empir Software Eng*, vol. 23, no. 6, pp. 3503–3534, Dec. 2018, doi: 10.1007/s10664-018-9613-x.

[9] R. Ollila, N. Makitalo, and T. Mikkonen, ‘Modern Web Frameworks: A Comparison of Rendering Performance’, *JOURNAL OF WEB ENGINEERING*, vol. 21, no. 3, pp. 789–813, 2022, doi: 10.13052/jwe1540-9589.21311.

[10] W3Schools, What is the DOM?

https://www.w3schools.com/js/js_htmldom.asp

[11] https://npmcompare.com/compare/react-native,svelte-native

[12] Dominic Burfrod – Medium: „Using the MVVM pattern with a Xamarin Forms mobile app“

https://domburf.medium.com/using-the-mvvm-pattern-with-a-xamarin-forms-mobile-app-b1930976bc3d

Skraćenice

IOS	iPhone Operating System
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
SDE	Software Development Environment
RX	Reactive Extensions
JS	JavaScript
JSON	JavaScript Object Notation
MVVM	Model-View-ViewModel
GUI	Graphical User Interface
DOM	Document Object Model
vDOM	virtual Document Object Model
API	Application Programming Interface
JSX	JavaScript XML
XML	Extensible Markup Language
RN	React Native
SN	Svelte Native
NS	Native Script