Upotreba Typescript programskog jezika za razvoj web aplikacija

Cindrić, Josip Ante

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

ZAVRŠNI RAD

UPOTREBA TYPESCRIPT PROGRAMSKOG JEZIKA ZA RAZVOJ WEB APLIKACIJA

Josip Ante Cindrić

Split, rujan 2021.

Temeljna dokumentacijska kartica

Završni rad

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

UPOTREBA TYPESCRIPT PROGRAMSKOG JEZIKA ZA RAZVOJ WEB APLIKACIJA

Josip Ante Cindrić

SAŽETAK

U ovom radu izložen je povijesni i teorijski kontekst programskih jezika JavaScript i Typescript te sličnosti i razlike oba jezika. Oba jezika su stavljena u kontekst razvoja Web aplikacija te su na praktičnom primjeru prikazane mogućnosti oba jezika uz pomoć React eksterne biblioteke te Firebase „Backend-as-a-Service“ platforme.

Ključne riječi: Web aplikacija, Typescript, JavaScript, biblioteka, React, Firebase, BaaC

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

Rad sadrži: 32 stranice, 9 grafičkih prikaza, 0 tablica i 21 literaturnih navoda. Izvornik je na hrvatskom jeziku.

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

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

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

Dr. sc. Divna Krpan, *viši predavač Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Rad prihvaćen: rujan 2021.

Basic documentation card

Thesis

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

USE OF TYPESCRIPT PROGRAMMING LANGUAGE FOR WEB APPLICATION DEVELOPMENT

Josip Ante Cindrić

ABSTRACT

In this thesis programming languages JavaScript and Typescript have been examined and given a historical and a theoretical context. Practical differences have been explained as well as the means of developing a Web application. The capabilities of both languages have been shown through the Web application built for this project. The project used React, a JavaScript library and Firebase, a "Backend-as-a-Service" platform.

Key words: Web application, Typescript, JavaScript, library, React, Firebase, BaaC

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

Thesis consists of: 32 pages, 9 figures, 0 tables and 21 references

Original language: Croatian

Mentor: **Goran Zaharija, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*

Reviewers: **Goran Zaharija, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*

Saša Mladenović, Ph.D. *Associate Professor of Faculty of Science, University of Split*

Divna Krpan, Ph.D. *Senior Lecturer of Faculty of Science, University of Split*

Thesis accepted: September 2021.

Uvod 2
1. Povijesni kontekst 3
1.1. Začetak Weba 3
1.2. Osnove Web tehnologija – HTML, CSS, JavaScript 3
1.3. Standardizacija JavaScripta 4
1.4. Kontekst nastanka Typescripta 6
2. Kontekst programskih jezika u domeni teorije programskih jezika 8
2.1. JavaScript 8
2.2. Typescript 12
3. Sličnosti i razlike JavaScripta i Typescripta 14
3.1. Zaključivanje tipova 14
3.2. Definiranje tipova 14
3.3. Kompozicija složenih tipova 15
3.4. Strukturalni sistem tipova 17
4. Kontekst razvoja Web aplikacija 20
5. Implementacija Web aplikacije koristeći Typescript 22
5.1. Korištene tehnologije 22
5.2. Opis aplikacije 23
5.3. Razlike u implementacijama JavaScript i Typescript dijelova projekta 23
5.4. Razlike u implementacijama komponenti 24
5.5. Izgled aplikacije 27
6. Zaključak 29
7. Literatura 30
8. Popis slika 32
9. Popis primjera koda 33

Uvod

Jedna od tri temeljne tehnologije na kojoj se zasniva Web kakvog danas poznajemo je JavaScript, programski jezik koji je ujedno i tema ovog rada uz njegov nadskup Typescript. Upitnik za 2021. godinu portala Stack Overflow programerima navodi JavaScript kao najkorišteniju tehnologiju sa znatnih 64.96% korisnika od 83.052 ispitanika, dok se na šestome mjestu sa 30.19% nalazi Typescript, programski jezik star, do trenutka pisanja ovog rada, samo osam godina.[0] Taj statistički podatak je potvrda da Typescript zaslužuje svu pažnju koju dobiva zadnjih nekoliko godina te je to jedan od razloga zašto je ova tema danas relevantnija no ikad.

Temeljni cilj ovog rada je usporediti programski jezik JavaScript te programski jezik Typescript u razvoju Web aplikacija. Proces kojim ćemo postići taj cilj obuhvaća izlaganje povijesti i konteksta nastanka svakog jezika te izlaganje konteksta oba jezika u sferi teorije programskih jezika, objektivno i egzaktno izlaganje sličnosti i razlika dva navedena jezika, razlaganje konteksta u kojemu se danas koriste i nalaze te na primjeru izgrađenom za završni informatički projekt prikazati mogućnost korištenja oba jezika za izgradnju Web aplikacije.

1. Povijesni kontekst

1.1. Začetak Weba

Informacijski sustav znan kao World Wide Web ili samo Web izumljen je 1990. godine od strane Tima Berners-Leea, znanstvenika tada zaposlenog u Švicarskom CERN-u u svrhu automatiziranja podjele informacija među tadašnjim znanstvenicima diljem svjetskih sveučilišta.[1] Slično kao i Internet dva desetljeća prije Web se ubrzo pokazao kao tehnologija korisna ne samo znanstvenicima već i široj populaciji. 1990. godine Tima Berners-Lee je razvio prvi Web preglednik te je sljedeće godine Web postao dostupan široj javnosti van CERN-a.[2] Web preglednici pri začetku tehnologije su za današnje standarde izgledali dosta primitivno. Glavni nedostatak ranih Web preglednika je bio manjak grafičkog sučelja. Taj problem je 1993. riješio Web preglednik imenom Mosaic. Bio je to prvi Web preglednik pristupačan korisnicima bez tehničkog obrazovanja potrebnog za baratanje prethodnicima Mosaica. Već je sljedeće godine bio zamijenjen sa novom poliranijom verzijom pod imenom Netscape Navigator koji je ubrzo postao tada najkorišteniji Web preglednik. Tvrtka Netscape zaslužna za Netscape Navigatora ubrzo je uvidjela potrebu za napretkom tehnologije na kojoj je izgrađen tadašnji web.

1.2. Osnove Web tehnologija – HTML, CSS, JavaScript

Današnji Web se enormno razlikuje od stanja u kojem se Web nalazio u njegovim začecima. Tehnologija koja opisuje konstrukciju Web stranice koju danas znamo kao HTML doživjela je prvi javni opis krajem 1991. godine, gotovo 2 godine nakon prve pojave Weba.[3] Taj inicijalni opis standardizirao je 18 elemenata koji su sačinjavali poprilično jednostavan sustav koji je dozvoljavao Web stranicama prikazivanje jednostavnih elemenata poput teksta, popisa, adresa i naslova što je i danas osnova modernog HTML5 standarda. CSS je kao tehnologija prvi put predložena krajem 1994. godine, a službeni standard je izdan 1996. od strane World Wide Web Consortiuma.[4] Netscape je 1995. pokrenuo razvoj posebnog programskog jezika namijenjenog implementaciji dinamičke funkcionalnosti Web stranicama uslijed propalih planova za korištenje programskog jezika Java te modifikacije Scheme programskog jezika u jezik sintakse više nalik Javi. Osoba odgovorna za ovaj poduhvat je Brendan Eich kojemu je

trebalo 10 dana da razvije prvu inačicu programskog jezika kojeg danas nazivamo JavaScript. U prosincu 1995. godine izdana je prva službena verzija JavaScripta, jezika koji će donijeti dinamičku funkcionalnost Webu te biti jedan od vodećih čimbenika u pokretanju revolucije informacijskog doba.[5] Brendan Eich je vremensko ograničenje koje mu je dano iskoristio za kreiranje multiparadigmatskog lightweight jezika kojemu će programeri budućnosti moći pristupit na različite načine bez mnogo restrikcija.

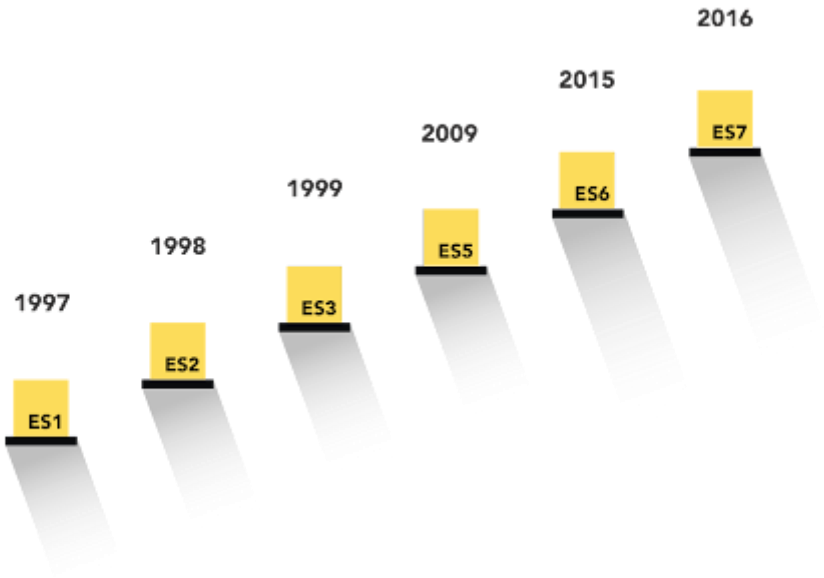
1.3. Standardizacija JavaScripta

Iste godine je Microsoft na tržište izbacio prvi vlastiti Web preglednik imenom Internet Explorer u pokušaju narušavanja monopola koji je Netscape pokušao uspostaviti. Sljedeće godine Microsoft je izdao JScript kao odgovor na JavaScript.[6] Microsoft je JScript razvio obrnutim inženjeringom Netscapeove implementacije JavaScripta. Razlike u implementacijama JScripta i JavaScripta zahtijevale signaliziranje korisnicima kojem Web pregledniku je namijenjena Web stranica za bezbolno surfanje.

Slika 1 Ikone signalizacije prikladnog preglednika za surfanje Web stranicom

Naredne godine Netscape je prijavio JavaScript kao predložak standardne specifikacije za sve Web preglednike organizaciji Ecma International što je rezultiralo standardizacijom ECMAScript jezične specifikacije 1997. godine.[7] Standardizacija JavaScripta u ECMAScript namijenjena je osiguravanju mogućnosti međusobnog funkcioniranja koda kroz različite Web preglednike. Sljedeće dvije godine ECMAScript je ažuriran u obliku novih izdanja nazivanih kraticama ES2 i ES3. Nova izdanja donijela su nove mogućnosti jeziku, ali uslijed političko ekonomske situacije napredak ECMAScript standarda naglo će stagnirati. Sljedeće izdanje ES4 ECMAScript standarda odbačeno je pri posljednjoj evaluaciji 2003. godine. Razlog takvom razvoju situacije je neslaganje osoba u odboru oko napretka jezika i implementacije novih mogućnosti koje su tada navedene kao previše kompleksne za jezik u cjelini. Naknadno su neke

preporučene mogućnosti implementirane u ES6 standardu 2015. godine, dok je ostatak mogućnosti vodilo ka implementaciji jezika sličnome Typescriptu. Rješenje već navedenog problema izrade Web aplikacija koje jednako funkcioniraju na svim Web preglednicima se počelo nagoviještati početkom druge polovice novoga desetljeća razvijanjem novih tehnologija kao što su Ajax, jQuery i node.js. Ajax što je skraćenica od „Asynchronous JavaScript and XML“ je koncept u programiranju to jest skup tehnika koje se koriste na strani klijenta za kreaciju asinkronih Web aplikacija. jQuery je biblioteka otvorenog koda dizajnirana sa namjerom pojednostavljivanja interakcije sa DOM-om, rukovanja događajima, CSS animacija te Ajax tehnika. Razlog uspjeha jQuery biblioteke je činjenica je to bila prva biblioteka koja je imala API koji je funkcionirao dobro na većini Web preglednika. Dodatna stavka je kvalitetno odrađena dokumentacija kojom se rijetko koja biblioteka tada mogla pohvaliti. 2008. godine izdan je Google Chrome Web preglednik i sa njime V8 JavaScript izvršni stroj(eng. engine). V8 izvršni stroj je program koji izvršava JavaScript kod sa jednom inovacijom. Do prvog pojavljivanja V8 izvršnog stroja JavaScript izvršni strojevi su JavaScript kod interpretirali, dok je V8 izvršni stroj implementirao novinu zvanu JIT kompilacija. Ova mogućnost uvela je revoluciju u JavaScript izvršne strojeve od kojih svi aktualni relevantni izvršni strojevi koriste ovu metodu izvršavanja koda. V8 izvršni stroj nije samo pokrenuo revoluciju u Web preglednicima već i na strani poslužitelja pojavom node.js izvršnog okruženja koje je omogućilo razvijanje aplikacija koje koriste JavaScript na strani poslužitelja i na strani klijenta. Sljedeće izdanje ECMAScript standarda ES5 izdano je 2009. godine te je donijelo mnoge nove mogućnosti te je otvorilo put kreaciji novih programskih okvira za izgradnju „Single page“ aplikacija to jest skraćeno SPA.



Slika 2 Razvoj ECMAScript standarda od ES1 od ES7

Novo izdanje ECMAScript standarda pojavilo se 2015. godine te je kao i prošlo izdanje donijelo puno novih mogućnosti, ali je time prouzročilo potrebu za programima koji će prevoditi moderan kod noviji ECMAScript izdanja u kod starijih izdanja pošto svi Web preglednici ne podržavaju najnovije mogućnosti. Takvi programi se zovu transkompilatori(eng. transcompiler) od kojih su najpoznatiji Babel i Typescript koji je u principu puno više od transkompilatora, ali može služiti toj svrsi. Također nakon ES6 standarda na tržištu su se pojavile mnoge nove tehnologije koje su omogućile jednostavnije razvijanje kompleksnih Web aplikacija kao što su novi SPA programski okviri, programi za upravljanje uvjetnim paketima te tehnologije koje dodaju statički sistem tipova JavaScriptu te mnoge druge inovacije.

1.4. Kontekst nastanka Typescripta

Typescriptovo prvo javno izdanje je izdano javnosti 2012. godine nakon dvogodišnjeg razvojnog perioda iza zatvorenih vrata Microsofta. Typescript je razvijen nakon ES5 standarda s namjerom olakšavanja JavaScript programerima razvoja aplikacija velikih razmjera. Glavna mana JavaScripta u aspektu razvijanja Web aplikacija velikih razmjera je nedostatak statičkog sistema tipova koji otežava razvijanje velikih modularnih aplikacija na kojima radi veliki broj timova. Još jedna bitna mogućnost Typescripta je već spomenuta transkompilacija u ranije

inačice ECMAScript standarda počinjući od ES3 standarda koja je u kombinaciji sa ostalim mogućnostima jezika iznimno korisna programerima.

2. Kontekst programskih jezika u domeni teorije programskih jezika

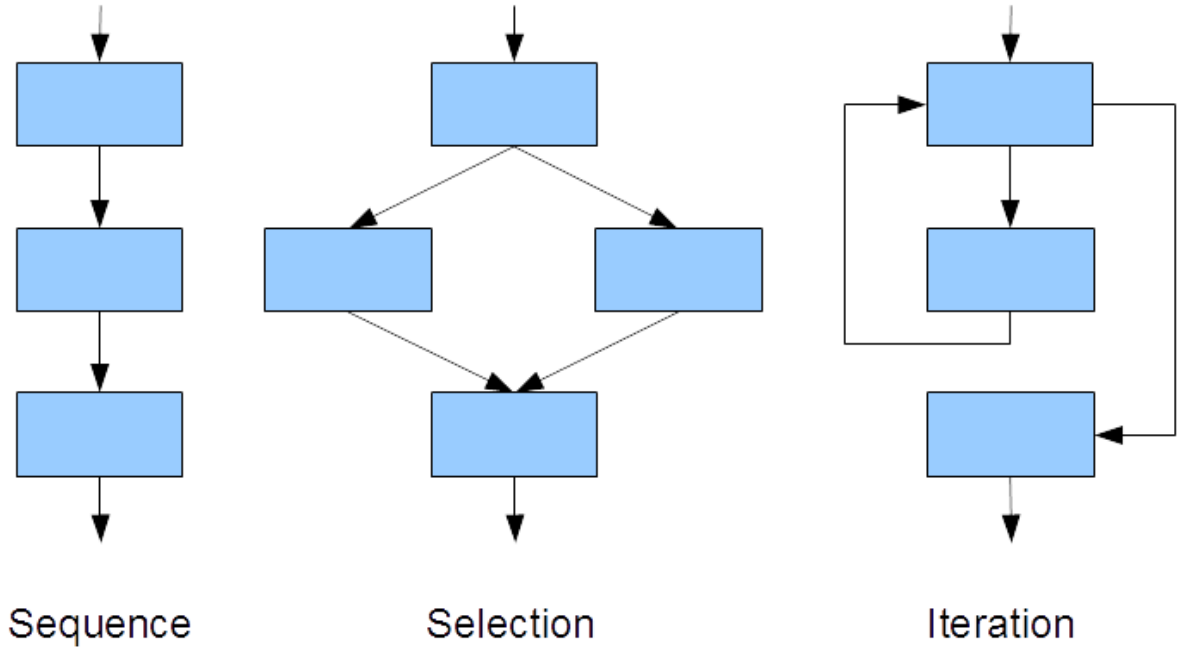
2.1. JavaScript

Kako bi pravilno razmotrili kontekst razvoja JavaScripta važno je razmotriti karakteristike jezika, pripadne paradigme u koje jezik spada te njegovo mjesto u široj teoriji programskih jezika. U ovoj analizi konteksta sagledat ćemo programske jezike treće generacije to jest programske jezike visoke razine zbog razine apstrakcije na kojoj se oni nalaze. Naime, programski jezici visoke razine nadovezuju se na asemblerske jezike te time omogućavaju programerima bavljenje isključivo konceptualno rješavanjem problema bez potrebe za detaljnim upravljanje rada procesora. Načini na koje možemo relevantno klasificirati programske jezike visoke razine jesu kroz programske paradigme i domene upotrebe.

Prema domenama upotrebe jezike možemo odjeliti u dvije kategorije, jezike namijenjene specifičnim domenama i generalno upotrebljive jezike. Domene programiranja odnose se na specifične rezultate koje proces programiranja luči poput programiranja video igara ili izvršavanjem na Webu kojoj je JavaScript inicijalno isključivo pripadao dok se danas može koristiti i kao jezik za programiranje aplikacija na strani poslužitelja pomoću Node.js izvršnog okruženja. Za razliku od navedenih jezika generalno upotrebljivi programski jezici se mogu primijeniti u različitim domenama, a to su jezici C, C++, C#, Go, Java, Python kao i mnogi drugi.

Druga metoda klasifikacije je malo kompleksnija. Programske paradigme klasificiraju programske jezike prema značajkama i većina programskih jezika visoke razine je multiparadigmatska od čega ni sam JavaScript ne odudara. Glavna programska paradigma većine današnjih programskih jezika je paradigma strukturiranog programiranja čija je glavna osobina naglasak na strukturiranju programa u obliku toka strukturnih kontrola, podrutina i blokova. Strukturne kontrole dijelimo na: kontrolu sekvence; naredbe ili podrutine izvršavane slijedom, kontrolu selekcije; naredbe se izvršavaju ovisno o stanju u kojemu se program nalazi, kontrolu iteracije; blok naredbi ponavljamo dok se ne zadovolji određeno stanje, kontrolu rekurzije; naredba poziva sama sebe dok se uvjet prekida ne zadovolji. Podrutine su jedinice koda sa mogućnošću poziva što omogućava referiranje prema sekvencama naredbi jednom naredbom. Slično podrutinama, blokovi omogućavaju odnos prema grupi naredbi kao prema

jednoj naredbi. JavaScript sadrži sve ove elemente što ga čini strukturalnim programskim jezikom.



Slika 3 Prikazi ilustracija struktura za dijagram toka

Sljedeća bitna osobina koja nema jedinstvenu programsku paradigmu, ali je iznimno bitna za klasifikaciju programskih jezika, je sistematizacija tipova. Sistem tipova je logički sistem koji dodjeljuje svojstvo tipa različitim programskim konstruktima. Osnovni cilj sistema tipova je eliminacija pojavljivanja bugova u kodu. Provjera ispravnosti tipova može se provjeravati statički ili dinamički to jest tijekom kompiliranja koda(eng. compile time) ili tijekom izvršavanja koda(eng. run time). Navedene faze dijelovi su životnog ciklusa programa. Vrijeme kompiliranja je faza u životnom ciklusu programa kod jezika koji se prevode jezik posrednik(eng. intermediate language) te se on izvršava tijekom faze izvršavanja koda umjesto izvornog koda kojeg je napisao programer. Na primjer JavaScript kod se u izvornim inačicama nije kompilirao već se direktno izvršavao to jest interpretirao dok se u novijim inačicama može izvršavati „Just-in-time“ ili JIT kompilacija koda koja efektivno kombinira kompilaciju i interpretaciju programa u strojni jezik tijekom procesa izvršavanja. Ove karakteristike JavaScript svejedno svrstavaju u paradigmu dinamičnih jezika kao i većinu ostalih skriptnih jezika. Druga bitna karakteristika sistema tipova koja proizlazi iz navedene sistematizacije je snažna to jest slaba implikacija tipova(eng. weak and strong typing). Teorijska analiza ove sistematizacije nadilazi mogućnosti ovoga rada svojom kompleksnošću pošto ne postoji konsenzus za egzaktne tehničke definicije ove karakterizacije. Valja spomenuti kako jezici snažne implikacije tipova provode stroža pravila pri implikaciji tipova varijabli ili vrijednosti,

što pri kompilaciji, što pri izvršavanju programa kod dinamičkih jezika. Jezici slabije implikacije tipova provode blaža pravila što može dovesti do nepredvidljivih ili ponekad čak i krivih rezultata pri izvršavanju programa ili jednostavno ima mogućnost implicitnog pretvaranja tipa tijekom izvršavanja koda. JavaScript je primjer jezika slabe implikacije tipova što je jedan od glavnih motivatora za razvitak Typescripta. Još jedna bitna osobina koja se pridodaje programskim jezicima je prolazak takozvanog pačjeg testa(eng. duck test). Ovaj test implicira prikladnost objekta za određenu oporabu uslijed njegovih svojstava i metoda za razliku od donošenja te odluke prema tipu toga objekta. Pačji sistem ćemo prikazati na sljedećem primjeru:

```
function Macka (ime) {
    this.ime = ime;
  }

  Macka.prototype.zvuk = function() {
    console.log('Mjau')
  }

  function Pas (ime) {
    this.ime = ime;
  }

  Pas.prototype.zvuk = function() {
    console.log('Vuf')
  }

  function proizvodiZvuk(zivotinja) {
    zivotinja.zvuk()
  }

  var mica = new Macka('Mica');
  var medo = new Pas('Medo');

  proizvodiZvuk(mica) // --> 'Vuf'
  proizvodiZvuk(medo) // --> 'Mjau'
```

Primjer koda 1 Primjer pačjeg sistema u JavaScriptu

Ovaj primjer pokazuje kako pačji sistem mari samo za postojanje funkcionalnosti koja se izvršava. Drugim riječima: „Ako nešto hoda kao patka i ako to nešto kvače kao patka, onda to mora biti patka“.[8] Prema mišljenju autora ovog rada ovakav opis može čitatelju dati krivu ideju o prirodi pačjeg sistema, pa ćemo to frazirati drugačije kako ne bi došlo do zabune u narednom poglavlju u kojem ćemo definirati strukturalni sistem tipova. Dakle: „Ako nešto može kvakati kao patka onda to nešto možemo smatrati patkom dovoljno da ga tražimo da radi

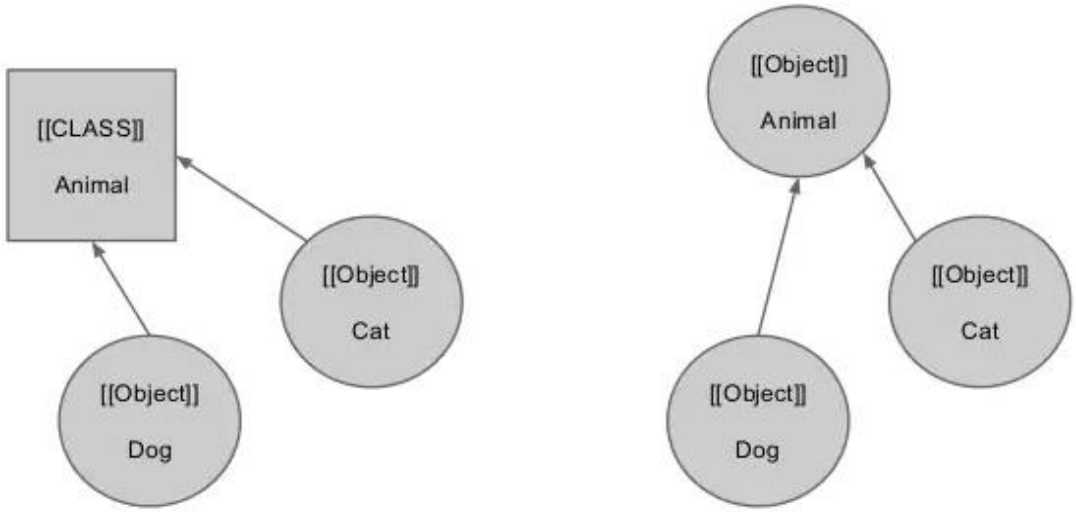
stvari koje bi mogla raditi patka“.[9] Iz primjera bi se po ovoj tvrdnji dalo zaključiti: „Ako neki objekt ima mogućnost proizvesti zvuk onda je za konstrukt koji to provjerava efektivno isti objekt“, a za dani primjer je taj konstrukt funkcija „proizvodiZvuk“.

Sljedeća bitna paradigma za razmotriti je paradigma imperativnog programiranja na koju se preko proceduralnog programiranja nadovezuje paradigma objektno orijentiranog programiranja koja se zasniva na konceptu objekta. Objekt je programski konstrukt koji sadržava podatke i ponašanje. Podatci su sadržani unutar dijelova objekta koje nazivamo svojstvima dok je ponašanje sadržano unutar procedura koje nazivamo metodama. Ideja imperativnog programiranja je da svaki program ima stanje koje se mijenja u ovisnosti o izvršenim naredbama. Direktna analogija na ljudski govor slijedi iz toga da imperativom u principu zapovijedamo to jest dajemo naredbe što je točno ono što radimo u imperativnom programiranju, dajemo naredbe računalu koje ih izvršava u obliku programa. Direktno na ideju imperativnog programiranja se nadovezuje proceduralno programiranje koje se zasniva na ideji pozivanja procedura koje su u suštini vrste podrutina. Procedure nisu ništa drugo negoli serija naredbi koje se izvršavaju jedna za drugom te imaju mogućnost izvršavanja pozivom jedne naredbe. Evidentno je kako je objektno orijentirano programiranje direktno nadovezano na proceduralno programiranje dodavanjem nove razine apstrakcije u obliku povezivanja podataka i funkcionalnosti to jest varijabli i procedura to jest svojstava i metoda u novi logički to jest programski konstrukt. Kako službeni opis JavaScript dokumentacije kaže „JavaScript podržava objektno orijentirani stil programiranja“.

Među ostalim „stilovima“ programiranja nalazimo suprotnost imperativnom programiranju u obliku deklarativne programske paradigme. Deklarativna programska paradigma opisuje stil programiranja koji od programera zahtijeva definiranje onoga što korisnik želi umjesto definiranja kako da se to postigne. Zajednička karakteristika većine programskih jezika ove paradigme je eliminacija korištenja primitivnih programskih konstrukta u svrhu ekspresivnosti kao što to na primjer biva u funkcionalnom programiranju. Teorijsko razlaganje ove paradigme također je izvan mogućnosti ovoga rada radi svoje duboke kompleksnosti, ali ćemo se zato kratko osvrnuti na funkcionalno programiranje koje proizlazi iz deklarativne ideje kao paradigma koja funkcije razmatra kao stabla naredbi koje preslikavaju dane vrijednosti unosa u konkretnu vrijednost izlaza što je kompatibilno sa matematičkom definicijom funkcije i ujedno u sukobu sa proceduralnom programskom paradigmom kojoj nije strano utjecati sa unutarnje strane procedure na vanjska stanja programa. Iz priloženih definicija teško bi bilo zaključiti da postoje programski jezici koji kombiniraju većine navedenih programskih

paradigmi pošto neke definicije daju naznake međusobnog isključivanja, ali sam JavaScript nam je primjer suprotnoga. Već spomenuta JavaScript dokumentacija navodi JavaScript kao programski jezik koji „podržava objektno orijentirano, imperativno i deklarativno to jest funkcionalno programiranje“.

Finalna tvrdnja prošlog paragrafa može implicirati različite zanimljivosti koje se kod JavaScripta konsolidiraju u programski stil koji je podvrsta objektno orijentirane paradigme imenom programiranje na bazi prototipova(eng. prototype-based programming). Njega definiramo kao stil u kojemu nasljeđivanje izvodimo putem „recikliranja“ postojećih objekata koji nam služe kao prototipovi za kreaciju novih objekata.



Slika 4 Shematski prikaz nasljeđivanja po klasi i po prototipu

Posljednja programska paradigma koje ćemo se dotaknuti je paradigma programiranja vođenog eventima(eng. event-driven programming). Ova paradigma nalaže da tok programa diktira pokretanje događaja kao što bi na primjer bili događaju uslijed korisnikove interakcije sa korisničkim sučeljem. Ova funkcionalnost je uobičajeno implementirana kroz beskonačnu petlju koja očekuje događaje(eng. event loop) te njima onda naknadno upravlja.

2.2. Typescript

Za razumjeti kontekst Typescripta kao programskog jezika bitno je znati da je on nadskup JavaScripta. To efektivno znači da je svaki validni JavaScript program zapravo i Typescript program koji će se transkompilirati točno sam u sebe ili u program izvršno ekvivalentnog rezultata napisan u jednom od ranijih ECMAScript standarda. Promjene koje Typescript uvodi naspram JavaScripta tiču se dvije distinktne sfere teorije programskih jezika.

Prva sfera je već spomenuta promjena u sistemu tipova gdje se uvodi promjena iz dinamičkog sistema tipova u postepen sistem tipova(eng. gradual typing). Ova tvrdnja se naizgled kosi sa već spomenutom promjenom iz dinamičkog u statički sistem tipova, ali problem leži u mogućnosti, a ne potrebi. Postepen sistem tipova omogućava programeru statičko definiranje tipova, ali ga na to ne prisiljava. Dakle programeru je ostavljen izbor odabira paradigme za određen dio programa ovisno o potrebi određene razine kompleksnosti. Druga promjena u istoj sferi je implementacija strukturalnog sistema tipova. Strukturalni sistem tipova implicira provjeru kompatibilnosti strukture koja je poslana i strukture koja se traži što ima sličan efekt kao pačji sistem, ali ta dva sistema funkcioniraju na različitim razinama. Strukturalni sistem tipova djeluje na razini kompilacije tj. prije transkompilacije dok pačji sistem djeluje pri izvršavanju JavaScript programa nakon transkompilacije Typescript programa. Efektivno ta dva sistema u određenim slučajevima djeluju sekvencijalno jedan za drugim što ćemo prikazati na primjeru u poglavlju 3.4.

Druga sfera nije direktna promjena koliko dodatak jedne programske paradigme. Riječ je o paradigmi generičkog programiranja koja omogućava pisanje algoritama koji dopuštaju naknadno definiranje tipova za tipove koji se definiraju kao parametri algoritma. Jednostavnije ova paradigma omogućava apstrahiranje tipova podataka u definicijama algoritama i strukturama podataka kako bi eventualnim korištenjem definiranih algoritama i strukturama definirali apstrahirano te time smanjili dupliciranje definicija za svaki mogući tip podatka u specifičnoj instanci algoritma ili strukture podataka. Primjer ove paradigme možemo vidjeti na primjeru koji prikazuje definiciju funkcije identitete matematički iskazana kao $f(x) = x$:

```
function identiteta<Type>(param: Type): Type {
    return param;
}

let mojaIdentiteta: <input>(param: input) => input = identiteta;
```

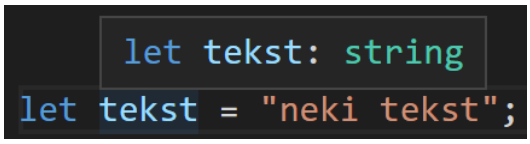
Primjer koda 2 Primjer generičke funkcije identitete

3. Sličnosti i razlike JavaScripta i Typescripta

U prijašnjim poglavljima već smo definirali odnos Typescripta i JavaScripta kroz aspekt paradigmi u koje jezike možemo svrstati. U ovome poglavlju valja razložiti sličnosti i razlike oba jezika što ćemo učiniti iz perspektive Typescripta kao vršitelja promjene. Veličina Typescript dokumentacije onemogućava razlaganje svih mogućnosti jezika u ovome radu, pa će ovo poglavlje pokriti samo osnove dovoljne za izradu aplikacije.

3.1. Zaključivanje tipova

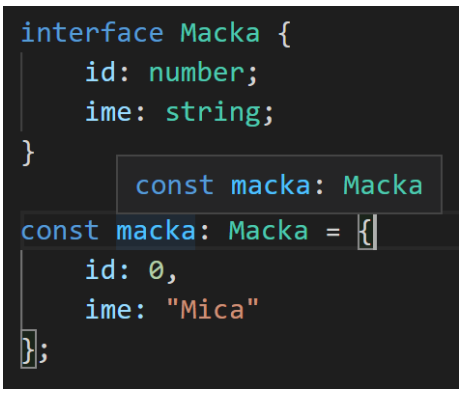
Prvi detalj koji je važno sagledati kod implementacije sistema tipova u Typescriptu je mogućnost Typescripta da zaključi o kojem tipu varijable se radi bez da tip specifično zadan. Na primjeru vidimo na koji način Visual Studio Code signalizira programeru o kojem se tipu varijable radi za dan podatak:



Slika 5 Primjer zaključivanja tipova

3.2. Definiranje tipova

Typescript implementira mogućnost statičkog zadavanja tipova zato što je u nekim uzorcima teško zaključiti o kojim se tipovima radi. U tim slučajevima programer može na prikladnim mjestima u kodu implicirati koji tip se pridodaje varijabli. Na primjer kod kreacije objekta možemo definirati koja svojstva objekt mora imati i koji su tipovi tih svojstava koristeći deklaraciju sučelja(eng. interface):



Slika 6 Primjer definiranja tipova

Na isti način možemo koristiti deklaraciju sučelja kod kreacije objekata korištenjem klasa i konstruktora te kod definiranja parametara i povratnih vrijednosti funkcija i metoda unutar tog sučelja. Također možemo definirati i opcionalne parametre i svojstva pri deklariranju istih što možemo vidjeti u sljedećem primjeru:

```
interface Macka {
    ime: string;
    starost:number;
    noviRodendan():void;
}

class MackaRodniList implements Macka{
    ime: string;
    starost: number = 0;

    constructor(ime: string, starost?: number) {
        this.ime = ime;
        if (starost !== undefined) {
            this.starost = starost
        }
    }

    public noviRodendan(): void {
        this.starost++;
    }
}

let maca: Macka = new MackaRodniList("Mica");
console.log(maca.starost); // 0
maca.noviRodendan()
console.log(maca.starost); // 1
maca = new MackaRodniList("Mica", 56);
console.log(maca.starost); // 56
```

Primjer koda 3 Definiranje tipova deklaracijom klasa

3.3. Kompozicija složenih tipova

Typescript također daje mogućnost programerima kreiranja kompleksnih tipova od kojih ćemo prvo spomenuti dva jednostavnija. Jedan način je korištenjem generika(eng. generic) ili drugim načinom korištenjem unija. Korištenjem unija programer *de facto* kreira skup mogućih tipova koristeći takozvani „pipe operator“. Izuzev povezivanja primitivnih i složenih tipova Typescript definira i tipove podataka koje naziva literalima(eng. literals) koji *per se* nisu pretjerano korisni, ali skup kreiran od više literala daju mogućnost programeru da opiše nove

korisne tipove koji su u principu unije podskupova primitivnih tipova. Na primjer možemo definirati literal:

```
type godisnjeDoba = "proljece" | "zima" | "jesen" | "ljeto";
```

Generici su programski konstrukti koji u principu kreiraju varijable tipova. Najjednostavniji primjer je prikaz polja koja mogu spremati određene tipove. Po definiciji polja spremaju podatke tipa „Any“ dok nam generici dopuštaju kreiranje tipa polja koji spremaju samo brojeve ili na primjer samo objekte tipa kojeg želimo spremiti:

```
interface Macka {
    ime: string;
    dob: number;
}

type PoljeBrojeva = Array<number>;
type PoljeObjekata = Array<Object>;
type PoljeMacaka = Array<Macka>;
```

Primjer koda 4 Primjer korištenja generika

S druge strane generike možemo koristiti za kreaciju novih složenih tipova. Na primjeru možemo vidjeti novi tip kreiran deklaracijom sučelja koji prima bilo koji tip pri deklaraciji i njega primjenjuje na instanciranu varijablu pri daljnjim pristupima:

```
class Macak {
    ime: string;

    constructor(ime: string) {
        this.ime = ime;
    }
}

interface Stablo<Tip> {
    naStablu: Tip;
    add: (obj: Tip) => void;
}

declare const stablo: Stablo<Macak>;
stablo.add(new Macak("Dipsi"));
stablo.naStablu = 45;
```

```
--> Type 'number' is not assignable to type 'Macak'.ts(2322)
```

Primjer koda 5 Kompleksniji primjer korištenja generika

Primjer na zadnjoj liniji daje grešku jer stablo ne dopušta pridjeljivanje podatka tipa broj svojstvu koje prima tip „Macak“.

3.4. Strukturalni sistem tipova

U poglavlju 2.2. definirali smo strukturalni sistem tipova, a na sljedećem primjeru prikazat ćemo na koji način Typescript zaključuje da su dva objekta ekvivalentna provjeravajući jednakost njihove strukture:

```
interface Zivotinja {
    ime: String;
    zvuk: () => void;
}

class Macka {
    ime: string;

    constructor(ime: string) {
        this.ime = ime;
    }

    zvuk():void {
        console.log("Mjau!");
    }
}

class Pas {
    ime: string;

    constructor(ime: string) {
        this.ime = ime;
    }

    zvuk():void {
        console.log("Vuf!");
    }
}

const proizvodiZvuk = (zivotinja: Zivotinja) => {
    zivotinja.zvuk();}
```

```
const mica = new Macka('Mica');
const medo = new Pas('Medo');
proizvodiZvuk(mica);
proizvodiZvuk(medo);
```

Primjer koda 6 Primjer strukturalnog sistema tipova

Strukturalni sistem tipova vrši provjeru u zadnje dvije linije kada provjerava ekvivalenciju između strukture poslanih objekata „mica“ i „medo“ i strukture objekta koje prima funkcija „proizvodiZvuk“. Na lebdjenje(eng. hover) akciju funkcija „proizvodiZvuk“ i objekt „mica“ daju ove definicije tipova:

```
const proizvodiZvuk: (zivotinja: Zivotinja) => void
const mica: Macka
```

Što dokaz djelovanja strukturalnog sistema. Transkompilacijom ovog primjera u ES5 standard efektivno dobijemo primjer pačjeg sistema kojeg smo prikazali u poglavlju 2.1. sa minimalnom izmjenom koda pri definiciji klase:

```
"use strict";

var Macka = /** @class */ (function () {
    function Macka(ime) {
        this.ime = ime;
    }
    Macka.prototype.zvuk = function () {
        console.log("Mjau!");
    };
    return Macka;
}());

var Pas = /** @class */ (function () {
    function Pas(ime) {
        this.ime = ime;
    }
    Pas.prototype.zvuk = function () {
        console.log("Vuf!");
    };
    return Pas;
}());

var proizvodiZvuk = function (zivotinja) {
    zivotinja.zvuk();
};
```

```
var mica = new Macka('Mica');
var medo = new Pas('Medo');
proizvodiZvuk(mica);
proizvodiZvuk(medo);
```

Primjer koda 7 Rezultat transkompilacije primjera koda 6

Primijetimo da ovo dokazuje pretpostavku iz poglavlja 2.2. koja kaže da su strukturalni sistem tipova i pačji sistem odvojeni sustavi koji djeluju sekvencijalno. Ako bi na isti primjer dodali objekt tipa „Stablo“ koje ima samo svojstvo „vrsta“, pri dodjeli objekta „stablo“ kao parametra funkciji „proizvediZvuk“ dobili bismo grešku:

```
interface Stablo {
    vrsta: string;
}

const stablo: Stablo = {vrsta: "bukva"}
proizvodiZvuk(stablo);
--> Argument of type 'Stablo' is not assignable to parameter of type
'Zivotinja'.
  Type 'Stablo' is missing the following properties from type
'Zivotinja': ime, zvuk ts(2345)
```

Primjer koda 8 Primjer greške uslijed strukturalnog sistema tipova

4. Kontekst razvoja Web aplikacija

Razvoju Web aplikacija prvo pristupamo odabirom tehnologije. Već spomenute tri tehnologije HTML, CSS i JavaScript sačinjavaju bazu na kojoj se grade Web stranice. Razlika između Web stranice i Web aplikacije u teoriji je vrlo jednostavna. Web stranica je kolekcija HTML, CSS i JavaScript datoteka koje sačinjavaju bez dinamičke funkcionalnosti. Web aplikacija sa druge strane uzima tu bazu te na nju dodaje dinamičku funkcionalnost. Ovisno o složenosti te dinamičke funkcionalnosti, veličine projekta i ostalim poslovnim uvjetima odabiru se tehnologije potrebne za razvoj aplikacije. Iako je moguće izgraditi Web aplikaciju samo sa HTML, CSS i JavaScript bazom rijetko tko će se danas odvažiti na takav poduhvat iz jedinstvenog razloga neisplate. Količina vremena potrebna za razvijanje Web aplikacije bez pomoći eksternih biblioteka ili razvojnih okvira je enormno veća za razliku od korištenja već definiranih „stackova“. „Stack“ je termin koji označava kolekciju tehnologija potrebnih za razvoj i plasiranje Web aplikacija. „Full stack“ je kolekcija tehnologija koja povezuje cijelu Web aplikaciju, pritom misleći na stranu klijenta(eng. front end) i stranu poslužitelja(eng. back end). Dio aplikacije na strani klijenta koristi već navedene tri tehnologije uz dodatak eksterne biblioteke ili razvojnog okvira za izgradnju korisničkog sučelja i dinamičke funkcionalnosti. Popularan izbor danas je odabir JavaScript eksternih biblioteka ili razvojnih okvira te CSS razvojnih okvira i procesora koji razvoju aplikacijskog dijela na strani klijenta čine iznimno jednostavnim za korištenje, a daju mnoga poboljšanja programerima od ubrzanja i poboljšanja kvalitete rada do dobre baze za ekspanziju aplikacije ako za to ima potrebe. Razvoj na strani poslužitelja može biti jednako bezbolan kao i razvoj na strani klijenta u ovisnosti o složenosti aplikacije. Arhitektura aplikacije na strani poslužitelja zasniva se na tehnologiji Hypertext Transfer Protokola koji omogućuje komunikaciju između instance Web aplikacije u Web pregledniku klijenta sa programom poslužiteljem na strani poslužitelja. Arhitektura Web aplikacije na strani poslužitelja se odvaja u četiri zasebna dijela: program poslužitelj, baza podataka, program posrednik(eng. middleware) te programsko sučelje aplikacije(eng. API). Ova arhitektura zahtjeva da svi ovi dijelovi postoje, ali ne nalaže da budu odvojeni. Napretkom tehnologije oblaka na tržištu su se pojavile platforme imenom „Backend-as-a-Service“ ili skraćeno BaaS koje programerima omogućavaju korištenje dijela aplikacije na strani poslužitelja kao servis bez potrebe za razvojem tog djela aplikacije, no kao potpuna suprotnost projekti mogu biti izgrađeni iz temelja ako je to zahtjev. U tom slučaju na softverskim

inženjerima je da izgrade taj dio aplikacije koristeći sve potrebne tehnologije. Na primjer, poznati „stackovi“ zasnivani na JavaScript tehnologiji imenom MERN(Mongo DB, Express.js, React, node.js) i MEAN(Mongo DB, Express.js, Angular, node.js) razlikuju se samo po razvojnom okviru na strani klijenta koristeći iste baze podataka, izvršna okruženja te kontrolnu razinu aplikacije. Primjer poznato „stackova“ koji ne koristi „JavaScript everywhere“ paradigmu bili bi LAMP(Linux, Apache, MySQL, PHP/Python/Perl) kojeg redom možemo razložiti na sumu operativnog sustava, programa poslužitelja, baze podataka i programskog jezika koji gradi Web aplikaciju. Dakle LAMP je samo inačica „solution stacka“ kojemu na tržištu ima dosta alternativa poput zamjene operacijskog sustava iz Linuxa u Windows ili Mac OS X, zamjene baze podataka iz MySQL u MariaDB, PostgreSQL, Oracle ili SQL Server, zamjene programa poslužitelja u na primjer NGINX ili ISS te zamjene programskog jezika drugim jezikom ili razvojnim okvirom.

5. Implementacija Web aplikacije koristeći Typescript

Kako bi se što bolje prikazale razlike između JavaScript i Typescript programskih jezika u praktičnom dijelu rada je implementirana jednostavna Web aplikacija koristeći oba programska jezika uz React biblioteku za dio aplikacije na strani klijenta dok je za aplikacijski dio na strani poslužitelja korišten BaaC sustav Firebase kako bi se minimizirale nepotrebne komplikacije.

5.1. Korištene tehnologije

Biblioteka React je besplatna JavaScript biblioteka otvorenog koda namijenjena dizajniranju korisničkih sučelja i njegovih komponenti. Srž Reactove funkcionalnosti je upravljanje stanjima aplikacije i prikazati to stanje DOMu što za kreiranje kompleksnih projekata zahtjeva korištenje dodatnih biblioteka za usmjeravanje ili dodatne funkcionalnosti za koje React nije specijaliziran. Osnovne građevne jedinica React projekata jesu komponente(eng. component) koje se mogu prikazati DOMu. Jedna od najbitnijih mogućnosti Reacta je korištenje takozvanog virtualnog DOMa. Virtualni DOM je kopija DOMa spremljena u memoriju na koju se vrše operacije React projekta nakon kojih se izračuna razlika Virutalnog DOMa i DOMa te se DOMu prikazuje samo razlika kako bi se spriječilo ponovno učitavanje Web stranice što svaku React aplikaciju čini SPA aplikacijom.

Firebase je platforma koja omogućava integraciju BaaC servisa aplikacijama. Integracijom Firebase projekta aplikaciji putem Firebase Software development kita, skraćeno SDKa, dopušta se toj aplikaciji korištenje servisa koje Firebase nudi koje redom glase:

- Authentication – sustav prijave korisnika
- Firestore database – NoSQL baza podataka, nova inačica
- Realtime database – NoSQL baza podataka, stara inačica
- Cloud storage – pohrana podataka u Oblaku
- Hosting – usluge poslužitelja
- Cloud functions – razvojni okvir bez potrebe za programom poslužiteljem
- Machine learning – SDK strojnog učenja za mobilne aplikacije

5.2. Opis aplikacije

Idejno aplikacija je dekonstrukcija online galerije slika u jednu stranicu sa elementima za pohranu i prikaz slika. Arhitekturno aplikacija se sastoji od dva dijela: React projekta i Firebase projekta. React projekt sačinjava klasična „Create React app“ struktura proširena organizacijom datoteka po tipu podataka. Struktura projekta izgleda ovako:[19]

```
+---public
\---src
    +---components
    +---firebase
    +---hooks
    \---res
```

Novo kreirane mape se nalaze unutar \src mape kako bi lakše organizirali React komponente, „hookove“, resurse i Firebase SDK konfiguracijske datoteke. Aplikacija i izgrađena od šest komponenti sljedeće hijerarhije:

Slika 7 Hijerarhija React komponenti

Firebase projekt je kreiran na Firebase portalu te su u svrhu aplikacije pokrenuti servisi „Cloud storage“ i „Firebase database“. Firebase projekt i React projekt su povezani pomoću konfiguracije unutar „\src\firebase\config.js“ datoteke putem čega omogućujemo React projektu korištenje spomenutih servisa bez potrebe za lokalnom pohranom istih.

5.3. Razlike u implementacijama JavaScript i Typescript dijelova projekta

HTML i CSS dijelovi React projekta su identični u obje implementacije dok su jedine razlike one u komponentama i hookovima. Hookovi u Reactu su funkcije odvojene u zasebne komponente koje omogućuju programerima manipulaciju stanjima i komponentama životnog ciklusa React aplikacija. Hookovi izrađeni u ovom projektu daju funkcionalnost komunikacije sa Firebase servisima. Prema tome su nazvani „useFirestore“ i „useStorage“. Pošto se tu ispod

površine ipak radi o React komponentama ekstenzije se među inačicama razlikuju kao .js za JavaScript implementaciju te kao .ts za Typescript implementaciju. Komponente koje smo vidjeli u hijerarhiji također se razlikuju po navedenim ekstenzijama osim što je u oba slučaja moguće iskoristiti .jsx i .tsx ekstenzije pošto se radi o izmjenično upotrebljivom formatu koji dodaje mogućnost ugrađene XML sintakse u klasične .js i .ts datoteke te se naknadno transkompilira u navedene formate.

5.4. Razlike u implementacijama komponenti

Funkcionalnost kroz sve komponente je identična. Jedina razlika je dodana implementacija sistema tipova u Typescript komponentama. Većina tipova je za programske konstrukte uključene iz eksternih biblioteka definirana što daje Typescriptu mogućnost zaključivanja kakvu smo prikazali u poglavlju 3.1. Prema tome na primjeru „App“ komponente možemo uvidjeti da je kod kroz obje implementacije isti.

```
import { useState } from 'react'
import Header from './Header'
import Gallery from './Gallery'
import Modal from './Modal'

const App = () => {
  const [selectedImg, setSelectedImg] = useState("");

  return (
    <div className="App">
      <Header />
      <Gallery setSelectedImg={setSelectedImg} />
      {selectedImg && <Modal selectedImg={selectedImg}
                             setSelectedImg={setSelectedImg} /> }
    </div>
  )
}

export default App
```

Primjer koda 9 App komponenta

Razlog tome je što „useState“ hook zaključuje pošto mu je poslan prazan string da treba kreirati state tipa string i prema tome prikladnu set funkciju tipa:

```
React.Dispatch<React.SetStateAction<string>>
```

Ako pogledamo komponentu „Gallery“ koja je odgovorna za kreaciju galerije slika u ovisnosti vraćenih podataka od strane „useFirestore“ hooka vidjet ćemo da su modifikacije za koje je odgovoran Typescript u principu potrebne samo za zadavanje tipa propuštenom svojstvu „setSelectedImg“ kroz React komponentu. Na primjeru također možemo primijetiti kako useFirestore hook daje Typescriptu do znanja tip podatka koji vraća prema čemu Typescript zaključuje koji tip treba zadati konstanti „docs“. Primijetimo također da tip koji je Typescript zadao konstanti docs nije definiran unutar komponente Gallery već hooka useFirestore.

```
import React from 'react'
import useFirestore from '../hooks/useFirestore';

//deklaracija ovog tipa ne postoji u .js verziji
type GalleryProps = {
    setSelectedImg: React.Dispatch<React.SetStateAction<string>>;
}

            // ({ setSelectedImg }) - .js verzija
const Gallery = ({ setSelectedImg }: GalleryProps) => {
    const { docs } = useFirestore('images');

    return (
        <div className="gallery">
            { docs && docs.map(doc => (
                <div className="img-
wrap" key={doc.id} onClick={() => setSelectedImg(doc.url)}>
                    <img src={doc.url} alt={doc.id}/>
                </div>
            ))}
        </div>
    )
}

export default Gallery;
```

Primjer koda 10 Gallery komponenta

Pogledajmo kako izgleda hook useFirestore. Ideja ovog hooka je u ovisnosti o primljenom imenu kolekcije vratiti sve podatke sadržane u toj kolekciji u obliku niza objekata tipa „Record“ kojeg ćemo deklarirati u hooku. U zaglavlju komponente smo unijeli potrebne metode iz eksternih biblioteka te tip „Timestamp“ iz eksterne biblioteke tipova Firestore servisa. Zatim smo kreirali strukturu objekta tipa Record deklaracijom sučelja kako bismo

mogli zadati pravilni tip kompozitnog objekta kojeg ćemo stvoriti od primljenog objekta tipa DocumentData iz kolekcije baze. Definirali smo parametar funkcije useFirestore kao string i povratnu vrijednost kao Record[] te smo unutar poziva baze podataka pristupili poslanim podacima i od njih kreirali niz Recorda koji nam i trebaju. U Typescript varijanti ovog hooka značenje koda je vrlo jasno, dok se u JavaScript verziji bez jasnih definicija tipova vrlo lako izgubi.

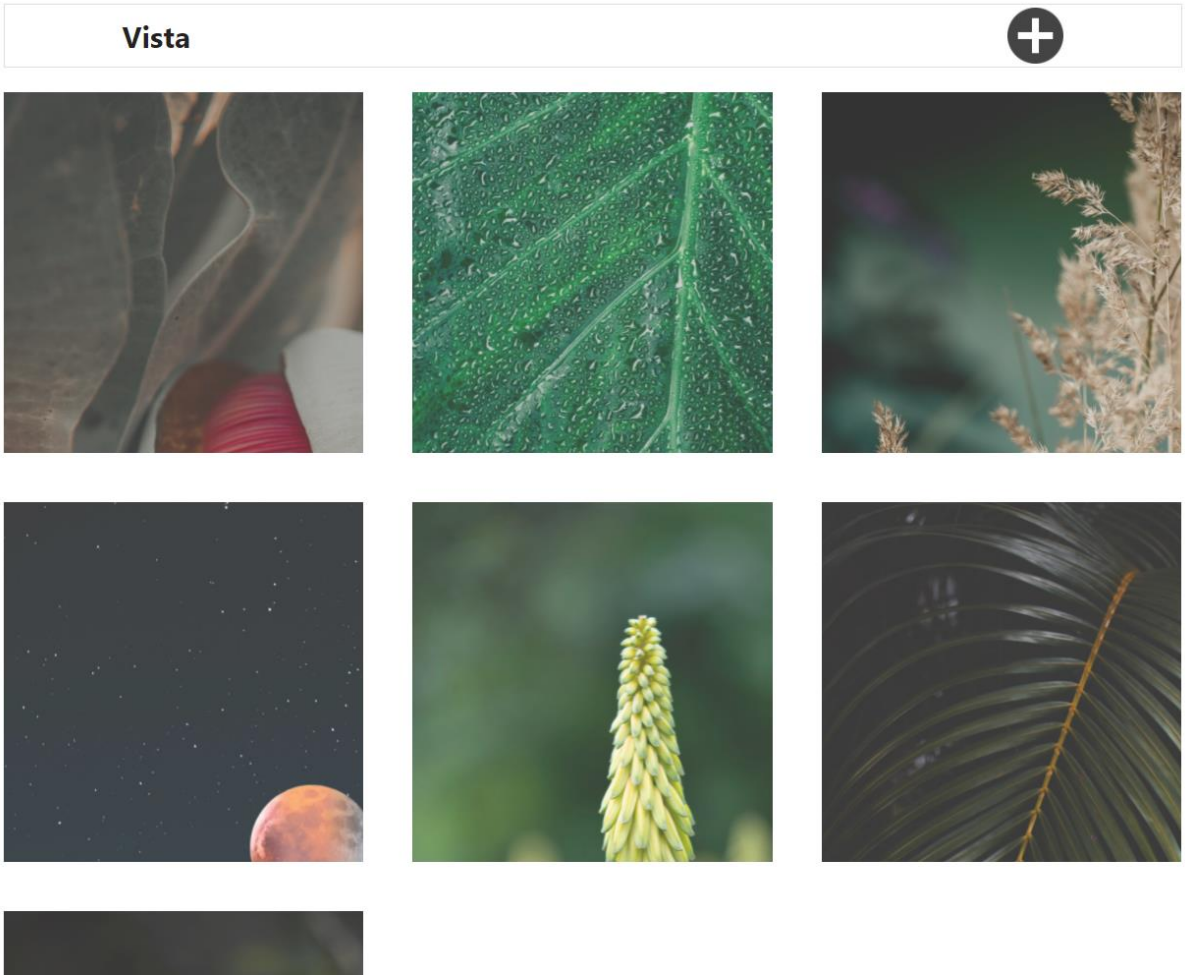
```
import { Timestamp } from '@firebase/firestore';
import { useState, useEffect } from 'react';
import { db, collection, onSnapshot, query, orderBy } from '../fireb
ase/config';

interface Record {
    id: string;
    timestamp: Timestamp;
    url: string;
}

const useFirestore = (coll: string) : Record[] => {
    const [docs, setDocs] = useState<Record[]>([]);

    useEffect(() => {
        const unsub = onSnapshot(query(collection(db, coll), orderBy
('timestamp', 'desc')), (snapshot) => {
            const documents: Record[] = [];
            snapshot.forEach((doc) => {
               documents.push({...doc.data(), id: doc.id} as Record)
            })
        setDocs(documents);
        });
        return () => unsub();
    }, [coll]);

    return docs;
}

export default useFirestore;
```

Primjer koda 11 Primjer useFirebase hooka

Ostatak komponenti i hookova funkcionira na jednakim principima.

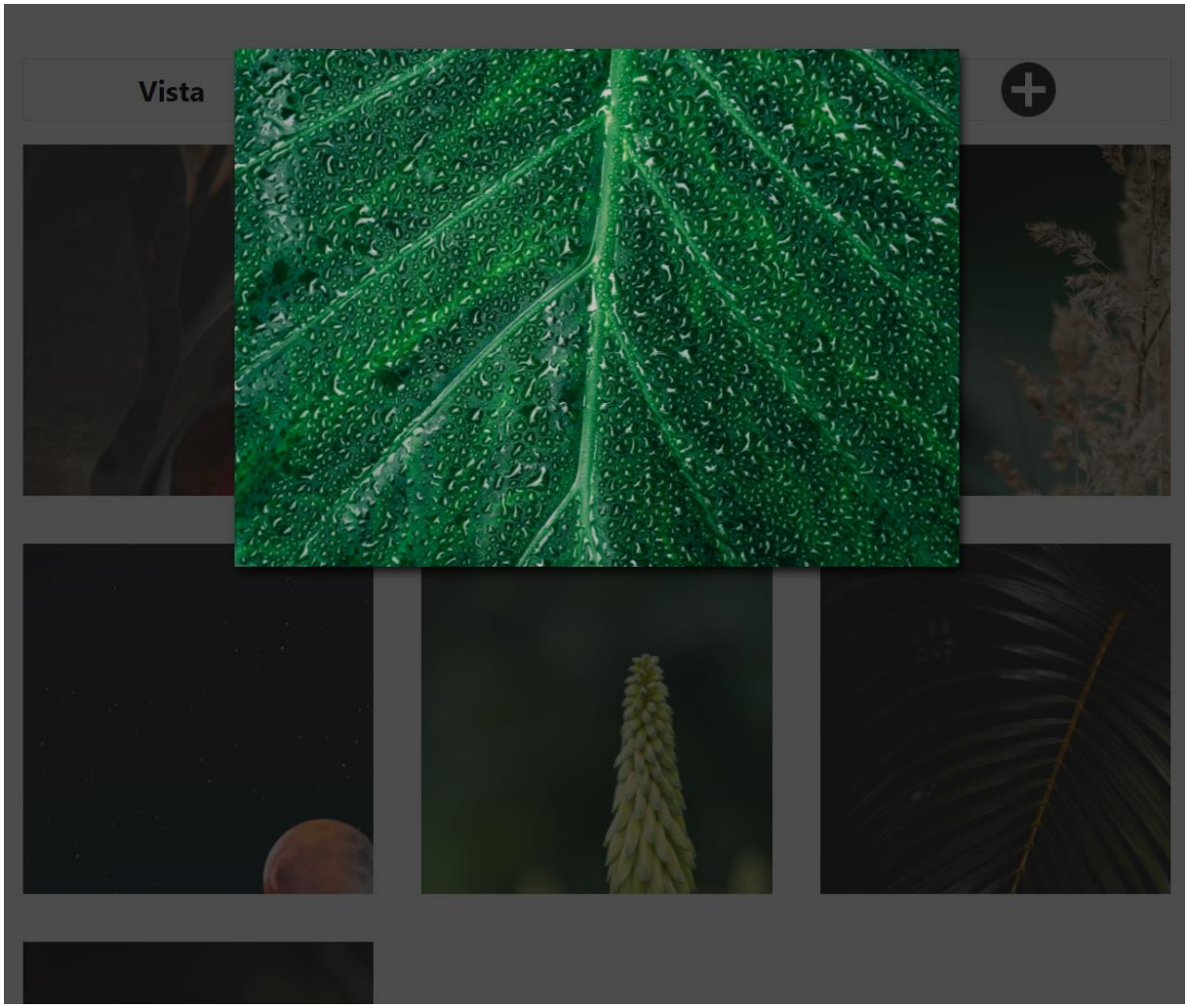
5.5. Izgled aplikacije

Prema prikazanoj hijerarhiji komponenti App komponenta je baza aplikacije koja sadrži sve ostale komponente. Od tri direktno sadržane komponente prema početnom stanju vidljive su samo dvije. Na događaj odabira jedne od slika u prvi plan dolazi prikaz komponente Modal koja uvećava prikaz odabrane slike. Početno stanje izgleda ovako:



Slika 8 Prikaz početnog stanja aplikacije

Popunjenost slika ovisi o aktualnom stanju baze podataka u trenu pokretanja aplikacije. Rezultat odabira slike iz galerije izgleda ovako:



Slika 9 Stanje stranice nakon odabira slike

6. Zaključak

U ovome radu izložili smo povijesni kontekst razvoja Weba i vezanih tehnologija, te smo sagledali JavaScript i Typescript u teorijskom aspektu i prikazali na koje se načine uklapaju u definirane programske paradigme i programerske stilove. Usporedili smo JavaScript i Typescript te prikazali promjene koje je Typescript uveo u razvoj aplikacija na Webu. Prikazali smo u koji kontekst spadaju JavaScript i Typescript u području razvoja Web aplikacija te smo na praktičnome projektu prikazali korištenje obje tehnologije za razvoj Web aplikacije uz pomoć React biblioteke.

Prema statistici navedenoj u uvodu Typescript nije popularan bez razloga. Na istome upitniku čak 72.73% od 24.909 ispitanika izrazilo je afekciju prema Typescriptu kao tehnologiji.[0] Samo su dva programska jezika rangirani bolje, Clojure sa 81.12% te Rust sa 86.98%, ali na više od 5 puta manjem uzorku. Kao dodatak obje tehnologije se koriste ne koriste u razvoju Web aplikacija. Za Typescript je budućnost jednako svijetla kao i za domenu Web aplikacija. Potražnja na tržištu za novijim, boljim i kvalitetnijim Web aplikacijama samo raste i malo je vjerojatno da će se to promijeniti uz tehnologije poput Typescripta.

7. Literatura

[0] 2021 Developer Survey, https://insights.stackoverflow.com/survey/2021

[1] WorldWideWeb: Proposal for a HyperText Project, https://www.w3.org/Proposal.html

[2] Tim Berners-Lee's original World Wide Web browser, http://info.cern.ch/NextBrowser.html

[3] HTML Tags, https://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html

[4] Cascading HTML style sheets -- a proposal, https://www.w3.org/People/howcome/p/cascade.html

[5] Chapter 4. How JavaScript Was Created, http://speakingjs.com/es5/ch04.html

[6] The Evolution of the Web Browsers, https://web.archive.org/web/20180831174847/https://www.mwdwebsites.com/nj-web-design-web-browsers.html

[7] Industry leaders to advance standardization of netscape's javascript at standards body meeting, https://web.archive.org/web/19981203070212/http://cgi.netscape.com/newsref/pr/newsrelease289.html

[8] Glossary, duck-typing, https://docs.python.org/3/glossary.html#term-duck-typing

[9] Jonathan Eunice Ph.D, 2014., Is there a difference between duck typing and structural typing? [duplicate], https://softwareengineering.stackexchange.com/questions/259943/is-there-a-difference-between-duck-typing-and-structural-typing

[10] JavaScript Documentation, https://developer.mozilla.org/en-US/docs/Web/JavaScript

[11] TypeScript Documentation, https://www.typescriptlang.org/docs/

[12] React Doumentation, https://reactjs.org/docs/getting-started.html

[13] Firebase Doumentation, https://firebase.google.com/docs

[14] Learning Duck Typing in Javascript, https://hackernoon.com/learning-duck-typing-in-javascript-qa3g35nc

[15] Microsoft Versus Netscape: The First Browser War, https://www.eyerys.com/articles/timeline/the-first-browser-war?page=5#event-a-href-articles-timeline-witness-tree-one-oldest-living-organisms-starts-tweeting039-a-witness-tree039-one-of-the-oldest-living-organisms-starts-tweeting-a

[16] Understanding ECMAScript and its new features,

http://blogs.quovantis.com/understanding-ecmascript-and-its-new-features/
[17] Logic, https://ict.senecacollege.ca/~btp100/pages/content/const.html
[18] JavaScript OOPs, https://www.slideshare.net/joujiahe/javascript-oo-ps
[19] Windows command tree output, https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/tree
[20] React Component Hierarchy Viewer output, https://www.npmjs.com/package/react-component-hierarchy
[21] Unsplash, https://unsplash.com/t/nature : Scott Webb, Gilberto Olimpio, Manny Moreno, Helen Ngoc N., Victoria Priessnitz, Khoa Ma, Tim Sessinghaus

8. Popis slika

Slika 1 Ikone signalizacije prikladnog preglednika za surfanje Web stranicom [15].....4

Slika 2 Razvoj ECMAScript standarda od ES1 od ES7 [16].....6

Slika 3 Prikazi ilustracija struktura za dijagram toka [17].....9

Slika 4 Shematski prikaz nasljeđivanja po klasi i po prototipu [18].....12

Slika 5 Primjer zaključivanja tipova.....14

Slika 6 Primjer definiranja tipova.....14

Slika 7 Hijerarhija React komponenti [20].....23

Slika 8 Prikaz početnog stanja aplikacije [21].....27

Slika 9 Stanje stranice nakon odabira slike [21].....28

9. Popis primjera koda

Primjer koda 1 Primjer pačjeg sistema u JavaScriptu[14] 10

Primjer koda 2 Primjer generičke funkcije identitete 13

Primjer koda 3 Definiranje tipova deklaracijom klasa 15

Primjer koda 4 Primjer korištenja generika 16

Primjer koda 5 Kompleksniji primjer korištenja generika 16

Primjer koda 6 Primjer strukturalnog sistema tipova 18

Primjer koda 7 Rezultat transkompilacije primjera koda 6 19

Primjer koda 8 Primjer greške uslijed strukturalnog sistema tipova 19

Primjer koda 9 App komponenta 24

Primjer koda 10 Gallery komponenta 25

Primjer koda 11 Primjer useFirebase hooka 26