

UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET

Primena statičke analize za generisanje koda jednostraničnih veb-aplikacija

MASTER RAD

Studijski program: xxx

Kandidat
dipl. inž. Lazar LJUBENović

Mentor
doc. dr Ivan PETKOVIĆ

Niš, 2018.

Zadatak:

Tralala

1. Doc dr. Ivan Petković
2. Prof dr. Suzana Stojković
- 3.

Datum prijave: _____
Datum predaje: _____
Datum odbrane: _____

Sažetak

Glavni cilj ovog rada je da demonstrira korišćenje statičke analize kao primarne tehnike za generisanje koda jednostaničnih aplikacija. Radi postizanja ovog cilja, razvijen je frejmwork Vejn (*Wane*), čije su osobine i mogućnosti detaljno opisane u ovom radu.

U radu je najpre izložen pregled osnovnih tehnologija koje se koriste prilikom izgradnje jednostraničnih aplikacija uz osvrt na prednosti i mane takvog pristupa spram veb-sajtova i desktop-aplikacija, kao i kratku istoriju veba kao platforme i Javaskripta kao jezika. Rad se zatim fokusira na Tajpskript; prvo se klasifikuje sistem tipova koji dolazi uz jezik i koji predstavlja njegovu najznačajniju prednost, a potom se, uz navođenje kratkih primera, detaljnije opisuju njegova svojstva. Posle toga je opisan izgled apstraktnog sintaksnog stabla koje odgovara kodu napisanom u Tajpskriptu, kao i način na koji mu se programski može pristupiti – prvo korišćenjem samog paketa `typescript`, a onda pomoću paketa `ts-simple-ast`, koji ujedno predstavlja i glavni alat koji se koristi u implementaciji frejmworka.

Zatim je pokazano da se sintaksno i semantički, iz ugla autora aplikacije, frejmwork po malo čemu izdvaja od postojećih rešenja koji se masovno koriste u industriji, kao što su *Angular* i *Vue*. Međutim, opisom načina rada ovog frejmworka, ističe se činjenica da je preciznije o Vejnu govoriti kao o kompilatoru, a ne frejmworku. Naime, izvorni kod se posmatra kao specifikacija ili definicija željenog ponašanja aplikacije i analizom istog se generiše kod specifičan za aplikaciju. Rad detaljno opisuje razne šablone koji se javljaju u kodu autorā i izlaže načine na koje se oni mogu prepoznati primenom tehnike analize toka kontrole koda, uzimajući pritom u obzir ne samo kod napisan u Tajpskriptu, već i definisan način interakcije elemenata aplikacije korišćenjem proizvoljne sintakse koja služi za deklaraciju šablona kojima se opisuje pogled komponenti.

Na kraju se poredi izlazni kôd jednostavne „todo” aplikacije napisane u nekoliko različitih frejmworka (*Angular*, *Vue*, *React*, *Preact*, *Mithril*, *Hyperapp*, *Wane*) kroz nekoliko metrika, među kojima su veličina koda i brzina izvršenja ažuriranja pogleda posebno istaknute.

Abstract

The main goal of this paper is to demonstrate usage of static analysis as primary technique for code generation of single-page applications. In order to achieve this, a framework named *Wane* has been developed, whose features and capabilities are herein described in detail.

The paper begins with an overview of basic technologies used for building single-page applications, along with advantages and disadvantages of such approach against websites and desktop applications, including a short history of web as a platform and JavaScript as a language. The focus then shifts on TypeScript; its most essential feature, the type system, is firstly classified, after which its characteristics are described in greater detail. After that, the structure of abstract syntax tree which corresponds to TypeScript code is described, as well as a way of accessing it programmatically – starting with the package `typescript` itself, and then leveraging `ts-simple-ast`, which is the main tool used to implement the framework.

It is then shown that, both syntactically and semantically, the framework is, from the author's perspective, not very different from the existing solutions massively used in the industry, such as Angular and Vue. However, after examining the way the framework works, it becomes evident that a more precise term for *Wane* is a compiler instead of a framework. Namely, the source code is viewed as a specification or a definition of the application's desired behavior, and by analyzing it, an application-specific code is generated. The paper describes different patterns found in authors' code and presents ways in which they can be recognized by utilizing technique of control flow analysis, taking into account not only code written in TypeScript, but also definition of the way elements of the application should interact through the means of custom syntax used to declare templates which describe components' view.

In the end, the built code of a simple “todo” application written in a few different frameworks (Angular, Vue, React, Preact, Mithril, Hyperapp, *Wane*) is compared using a few metrics, of which code size and view update time are of particular importance.

Sadržaj

1	Moderni veb	1
1.1	Jednostranične aplikacije	1
1.1.1	Tehnologije na koje se oslanja SPA	3
1.1.2	Prednosti	3
1.1.3	Mane	4
1.2	Sazrevanje Javaskripta	5
1.2.1	Nastanak	5
1.2.2	Pokušaji popravke	6
1.2.3	Godišnji ciklusi	6
1.2.4	Odbor TC39	7
1.3	Biblioteke i frejmvorci	8
1.3.1	Detekcija promena	10
1.4	Pomoćni alati i novi jezici	12
1.4.1	Moduli u Javaskriptu	12
1.4.2	Bandleri i minifikacija	13
1.4.3	Javaskript kao asembler veba	15
1.5	Vejn	16
2	Tajpskript	17
2.1	Klasifikacija sistema tipova	17
2.1.1	Podele	18
2.1.2	Deklaracije	19
2.1.3	Definisanje tipova	19
2.1.4	Strukturalni sistem tipova	20
2.2	Konfiguracija	20
2.2.1	Izbor ES verzije	21
2.2.2	Moduli	22
2.2.3	Globalni tipovi	22
2.2.4	Emitovanje koda	22
2.2.5	Lintanje koda	23
2.2.6	Strogi režim	24
2.3	Tipovi	24
2.3.1	Unija i presek	26

2.3.2	Interfejsi i klase	26
2.3.3	Osnovni tipovi	27
2.3.4	Čuvari tipova	28
2.3.5	Tip never	30
2.3.6	Preklapanje funkcija	30
2.3.7	Generički tipovi	31
2.3.8	Uslovni tipovi	32
2.3.9	Manipulacija tipovima	32
2.3.10	Primeri tipova iz lib	33
3	AST Tajpskripta	35
3.1	Tajpskriptov kompilator	35
3.2	ts-simple-ast	36
3.2.1	Vrste sintakse	36
4	Mogućnosti Vejna	39
4.1	Šabloni	39
4.1.1	Vezivna sintaksa	40
4.1.2	Interpolacija	40
4.1.3	HTML elementi	41
4.2	Komponente	42
4.2.1	Deklaracija	42
4.2.2	Pristupna komponenta	43
4.2.3	Korišćenje	43
4.2.4	Ulazi	43
4.2.5	Izlazi	44
4.3	Direktive	44
4.3.1	Uslovi	45
4.3.2	Nizovi	45
4.3.3	Skraćeni oblik	46
4.4	Stilovi	46
4.4.1	Enkapsulacija	46
4.4.2	Nadogradnja	47
4.4.3	SCSS	48
4.5	Pokretanje	48
4.5.1	Produkcija	48
4.5.2	Razviće	49
4.6	Primer aplikacije	49
5	Kompilacija Vejn aplikacije	51
5.1	Proces analize koda	51
5.1.1	Stablo analizatora fabrika	51
5.1.2	Analiza toka koda	53
5.1.3	Analizatori komponenti	55
5.1.4	Analizatori fabrika	56
5.2	Proces generisanja koda	60

5.2.1	Priprema	60
5.2.2	Preprocesiranje komponenti	60
5.2.3	Generisanje fabrika	62
5.2.4	Postprocesiranje komponenti	62
5.2.5	Bandlovanje	63
5.3	Anatomija fabrika komponenti	64
5.3.1	Metoda init	65
5.3.2	Metoda diff	70
5.3.3	Metoda update	71
5.3.4	Metoda destroy	71
5.4	Fabrike uslovne direktive (w:if)	72
5.4.1	Uslovni pogled	72
5.4.2	Parcijalni pogled	74
5.5	Fabrike direktive za ponavljanje (w:for)	75
5.6	Stilovi	76

Moderni veb

Samo hoću da prikažem podatke na stranici, ne da izvedem Sab-Zirov *fatality* iz originalnog MK.

„Kakav je osećaj učiti Javaskript u 2016”, Hose Aginaga [2]

Teško je utvrditi tačan trenutak kada je internet nastao, ali se vezuje za pojavu elektronskih računara pedesetih i prve konceptualne prototipe regionalnih računarskih mreža šezdesetih [35]. Ipak, internet kao mreža kakvu danas poznajemo vuče korene iz 1989, kada je **Tim Berners-Li** dobio ideju da sistem hipertekstualnih linkova, koji omogućuju skakanje sa jednog dokumenta na drugi, „samo poveže sa TCP-jem i DNS-om i — tada! — *World Wide Web*” [5].

Zatim je 1990. napisao prvu specifikaciju HTML-a koju je objavio na internet 1992. pod nazivom „HTML tagovi”, gde je naveo i opisao osamnaest tagova. Među njima su neki i danas prepoznatljivi, sa identičnim semantičkim značenjem: `title`, `a`, `p`, `h1` do `h6`, `address`, `dl`, `dt`, `dd`, `ul` i `li` [6].

Osim toga, razvio je prvi brauzer – *WorldWideWeb*, kasnije preimenovan u *Nexus*. Brauzer je istovremeno imao i ugrađen WYSIWYG editor [7].

Stvari su se potom odvijale brzo – brauzeri su se takmičili za popularnost i međusobno smenjivali; HTML je dobio zvaničnu specifikaciju, iz godine u godinu dobijajući nove standarde; pojavio se CSS kao jezik za definisanje stilova; a Javaskript postaje prvi programski jezik koji se izvršava u brauzeru.

Jednostranične aplikacije

Danas se većina aplikacija izvršava upravo u brauzeru, i polako ali sigurno potiskuju klasične desktop aplikacije iz upotrebe. Ovo ima brojne prednosti, ali se sledeće dve izdvajaju kao ključne.

Veb-aplikacije od korisnika **ne zahtevaju instalaciju**. Mnogo je veća verovatnoća da će potencijalni klijent koristiti aplikaciju ukoliko može da je koristi čim naiđe na nju, umesto da mora da je preuzme i instalira na svojoj mašini. Sem toga, danas nije retkost da korisnici poseduju više uređaja koje svakodnevno koriste. Veb-aplikacija je po definiciji dostupna sa svakog od njih, dok bi desktop ili mobilnu aplikaciju trebalo instalirati na svakom uređaju ponaosob.

Pored ovoga, **ažuriranje** klasičnih desktop aplikacija zahteva neki vid akcije od korisnika. Čak i ako je ažuriranje dobro isplanirano i može da se odvija automatski i bez čitave ponovne instalacije programa, korisnici uvek imaju mogućnost da ipak ostanu na staroj verziji. Developer ne može da bude siguran u to koju verziju aplikacije koriste klijenti, što za posledicu ima razne probleme u kompatibilnosti prilikom, na primer, komunikacije sa serverom, koji bi morao da uvek podržava sve verzije desktop aplikacije. Veb-aplikaciju je trivijalno ažurirati; garantovano je da će svi korisnici istog trenutka moći da pristupe jedino najnovijoj verziji.

Sadržaj na internetu je prvobitno zamišljen kao skup statičkih dokumenata koji su međusobno povezani. Praćenjem tih veza, korisnik pomoću brauzera sa svog računara (klijent) izdaje udaljenom i znatno moćnijem računaru (server) zahtev za pribavljanje nove stranice, koristeći URL kao identifikator. Kao odgovor na zahtev, server klijentu vraća novu stranicu, koju klijent iznova renderuje [5]. Za izgradnju ovakvih statičkih stranica dovoljno je koristiti HTML (*HyperText Markup Language*), kojim se definiše semantička struktura sadržaja stranice, i CSS (*Cascading Style Sheet*), koji služi za deklaraciju stilova, tj. načina prikaza (prezentacije) dokumenata pisanih u HTML-u.

Kada brauzer primi stranicu od servera, on je parsira. Na osnovu HTML-a kreira strukturu podataka u vidu stabla koja se naziva DOM (*Document Object Model*), a na osnovu CSS-a se na sličan način generiše CSSOM (*Cascading Style Sheets Object Model*). Na osnovu ova dva stabla brauzer može da renderuje stranicu [16].

Statičke veb-strance su vremenom zamenile dinamičke veb-stranice, koje su korisnicima omogućile određeni vid interakcije. Koreni ovog pristupa sežu u takozvanim „bogatim” internet aplikacijama (*Rich Internet Application*, RIA), koje su se oslanjale na tehnologije kao što su Java apleti (*Java Applets*), Adobejev Fleš (*Flash*) i Majkrosoftov Silverlajt (*Silverlight*). Bez korišćenja ovih tehnologija (koje se danas smatraju zastarelim jer ih brauzeri više ne podržavaju [24]), da bi stranica bila dinamička, osim HTML-a i CSS-a, mora se pisati i kod u Javaskriptu kojim se programira način interakcije sa korisnikom. Brauzeri nude API ka DOM-u koje programer može iskoristiti da osluškuje događaje (klik mišem, pritisak dirke na tastaturi, itd), i da programski manipuliše strukturom DOM-a. Ovo znači da stranica može promeniti svoju strukturu i izgled na osnovu akcija korisnika, bez potrebe da se sa servera dobavi cela nova stranica, i bez potrebe da je brauzer iznova parsira i renderuje.

Takav način ponašanja stranice se u početku koristio samo za neke manje izmene

na stranici koje je bilo trivijalno promeniti Javaskriptom, npr. za validaciju formi na klijentu: korisnik odmah nakon unosa lozinke može biti obavešten da ona mora imati više od određenog broja karaktera (umesto da pošalje formu, da se na serveru detektuje nevalidna lozinka, i da se vrati nova stranica sa upisanom greškom). Ovaj vid funkcionisanja predstavlja preteču veb-aplikacija, kod kojih se sve više funkcionalnosti događa na klijentu umesto na serveru.

Veb-aplikacija je klijentska aplikacija kod koje se sav korisnički interfejs izvršava u okviru brauzera. Ne postoji jasna granica između „dinamičke veb-stranice” i „veb-aplikacije”; za sajtove koji po izgledu više podsećaju na desktop ili mobilnu aplikaciju postoji veća šansa da se proglase veb-aplikacijama. Među veb-aplikacijama se posebno izdvajaju **jednostranične aplikacije** (SPA, *Single-Page Application*) jer se potpuno odvajaju od tipične veb-paradigme gde se korisnik kreće kroz stranice koje imaju različite URL-ove.

Nema preciznih podataka o tome kada se prvi put upotrebio termin „*single-page application*” u tom obliku, ali postoje članci koji pominju slične reči i taj koncept iz 2003. godine, mada ideju prvenstveno nazivaju „unutrašnje surfovanje” (*inner browsing*) [26]. Najraniji poznati primer jednostranične aplikacije je sajt `slashesdotslash.com`, iz aprila 2002. Razvio ju je Stjuart „stunix” Moris kao eksperiment u kome je demonstrirao funkcije i mogućnosti dinamičkog HTML-a, CSS animacija i JavaSkripta [53]. Najpoznatiji primeri prvih jednostraničnih aplikacija su *Gmail* (2004) i *Google Maps* (2005) [54].

Tehnologije na koje se oslanja SPA

U osnovi modernih SPA aplikacija leži **Ajaks** (AJAX). Striktno govoreći, ne radi se o tehnologiji, već o skupu više tehnologija, od kojih se kao najznačajnija izdvaja XMLHttpRequest koja u kombinaciji sa HTML-om, CSS-om, Javaskriptom, DOM-om i XML-om ili JSON-om omogućava inkrementalno ažuriranje korisničkog interfejsa bez slanja zahteva ka serveru za novu HTML stranicu, i samim tim bez osvežavanja čitave stranice. Na ovaj način je aplikacija znatno responzivnija, u smislu da brže odgovara na zahteve korisnika.

Mada je X u AJAX oznaka za XML, za razmenu podataka se relativno brzo sa XML-a prešlo na JSON, iz razloga što predstavlja kompaktniji zapis podataka, ali i zbog toga što od 2009. godine Javaskript ima ugrađenu funkciju za parsiranje JSON formata u Javaskript objekat.

Međutim, vremenom se pokazalo da je XMLHttpRequest nezgrapan i previše komplikovan za rad. Dolazak prototipa Promise u ES2015 iskorišćen je za definiciju novog standarda za obavljanje Ajaks zahteva pod nazivom `fetch`.

Prednosti

Kao prednosti jednostraničnih aplikacija izdvajaju se tri glavne osobine.

Jednostranične aplikacije su **brze**. Većina resursa se učitava samo jednom (lejaut, stilovi, skripte). Samo se podaci razmenjuju sa serverom. Na primer, u aplikacijama kao što su mejl-klijenti, struktura aplikacije ostaje ista tokom navigacije kroz nju. Umesto da se otvaranjem mejla iznova učitava čitava stranica, dobavlja se samo sadržaj mejla i zatim se Javaskriptom taj sadržaj ubacuje u odgovarajuću sekciju sajta – ostatak stranice nema potrebe da se ponovo preuzima sa servera.

Iskustvo korisnika je znatno bolje prilikom korišćenja jednostraničnih aplikacija spram klasičnih veb-sajtova. Ovo je direktna posledica goreopisane brzine. Ako se iskoriste neke tehnike za optimistično učitavanje podataka *pre* nego što ih korisnik zatraži, moguće je učiniti da se promene koje korisnik zahteva dogode (naizgled) istog trenutka.

Pošto jednostranične aplikacije konzumiraju **API samo za razmenu podataka**, identičan API se može iskoristiti i za druge aplikacije; na primer, mobilna aplikacija može pozivati iste tačke na API-ju, a interno na drugačiji način prikazivati te podatke. Ovo je nemoguće ostvariti u tradicionalnoj paradigmi jer server vraća čitavu HTML stranicu, koja je mobilnom klijentu beskorisna.

Mane

Mane jednostraničnih aplikacija su mnogo suptilnije i uglavnom se tiču činjenice da one ne mogu u potpunosti iskoristiti funkcionalnosti brauzera, već da moraju manuelno iznova implementirati te funkcionalnosti u Javaskriptu.

Brauzeri čuvaju **istoriju posećenih stranica** što čini povratak na prethodnu stranicu veoma brzim. Kada korisnik pritisne dugme za navigaciju unazad, očekuje da se promena dogodi posle veoma kratkog vremena i da stranica bude u sličnom stanju u odnosu na ono kada je poslednji put bila prikazana na ekranu.

Kada je sajt izgrađen tradicionalnim modelom, brauzer će biti u stanju da iskoristi keširanu verziju stranice i povezane resurse. Naivna implementacija jednostranične aplikacije će povratak nazad poistovetiti sa bilo kojom drugom akcijom, odnosno biće okinut događaj koji će detektovati osluškivač u Javaskriptu, što će kao posledicu imati ponovno ispaljivanje zahteva ka serveru za istim podacima; dakle javlja se dodatna latentnost i (verovatno) vizuelna promena.

Da bi korisnici mogli da imaju zadovoljavajuće iskustvo prilikom navigacije kroz jednostraničnu aplikaciju, slična funkcionalnost se mora implementirati u Javaskriptu. Aplikacija treba da kešira stranice korišćenjem memorije, lokalnog skladišta, baze podataka na strani klijenta ili kolačića. Sem toga, aplikacija mora da odluči i kada da pokupi te stranice i iskoristi ih. Da bi se ovo rešilo, neophodno je da se napravi razlika između praćenja veze (ili kucanja adrese direktno u brauzer) i pritiska na dugmad *back* i *forward* u brauzeru. Međutim, iz bezbednosnih razloga, nemoguće je iz Javaskripta ustanoviti na koji je način korisnik navigirao na drugu stranicu.

Sličan problem postoji i kod čuvanja **pozicije skrola**. Brauzeri pamte poziciju skrola na stranicama koje su ranije posećene. Pošto se kod jednostraničnih aplikacija stranica nikada ne menja, brauzer nije svestan promene „stranica”, pa čuva poziciju skrola čak i kada se to ne očekuje. Na primer, ako korisnik klikne na link na dnu stranice, kada mu se otvori „nova” stranica, brauzer neće znati da treba da ažurira poziciju skrola na vrh, pa će korisnik biti odveden na dno.

Ovo je će biti moguće delimično zaobići korišćenjem *Custom Scroll Restoration* API-ja. Međutim, u slučaju da se prethodno dobavljeni podaci ne keširaju, ili da postoji bilo kakvo kašćenje između otvaranje stranice i prikaza asinhrono podataka, pozicija skrola će i dalje biti pogrešna iz ugla korisnika.

Zatim, brauzeri će korisnike obavestiti da imaju **popunjenu formu** koja još uvek nije poslata kada probaju da navigiraju na neku drugu stranicu. Ovo je implementirano kroz događaj *beforeunload* na koji se developer može pretplatiti. Međutim, kod jednostraničnih aplikacija ovaj događaj postaje beskoristan jer brauzer ne pravi zahteve ka pravim stranicama. Developer bi morao lično da bude odgovoran da pruži ovakvo iskustvo korisnicima, što povećava kompleksnost koda.

Sazrevanje Javaskripta

Nastanak

Javaskript je nastao maja 1995. godine kao „jezik-lepak” [28] za tadašnji dominantni brauzer Netscape (Netscape) [49], kada ga je **Brendan Aik** definisao za samo deset dana. Prvobitno nazvan *Mocha*, četiri meseca kasnije biva prekršten u *LiveScript* da bi tri meseca nakon toga dobio ime po kojem je i danas poznat: *JavaScript* [3]. Ovaj jezik je nastao iz potrebe za premošćavanjem jaza između brauzera kao korisničkog okruženja za prikaz dokumenata i jezika za opis tih dokumenata, HTML-a.

Sa pojavom drugih brauzera, pojavili su se i jezici koji su očigledno bili inspirisani Javaskriptom, ali njihovi stvaratelji nisu smeli da ga zvanično tretiraju kao Javaskript zbog autorskih prava koji je imao San Majkrosistems (*Sun Microsystems*). Na primer, Javaskript je poslužio kao podloga Adobiju za jezik Ekšnskript (*ActionScript*), i sada predstavlja njegov zvaničan dijalekat. Ipak, najpoznatiji primer kopije Javaskripta je nesumnjivo *JScript* koji je došao uz Majkrosoftov Internet Explorer u verziji 3, već krajem 1996. Osim u imenu, razlikovao se i u nekim detaljima u implementaciji, ali i u API-ju za komunikaciju sa DOM-om. Iz tog razloga je ubrzo organizovana komisija za standardizaciju Javaskripta, nazvana ECMA [50].

Pokušaji popravke

Osnova za moderni Javaskript definisana je 1999. godine u trećoj verziji standarda, **ES3**. Počelo se sa radom na četvrtoj verziji, koja je trebalo da bude još radikalnija i da ispravi neke „greške” u Javaskript standardu. Najavljivane promene su bile toliko velike da su čak dovele do toga da ES4 bude postane poznat kao „Javaskript 2”. Međutim, komitet za standardizaciju je bio podeljen: dok se jedna strana zalagala za poboljšanje jezika zarad bolje budućnosti, druga se bojala da će ovime nastati prevelika pometnja, jer bi sve veb-stranice koje koriste „stari” Javaskript prestale da rade zbog promene u sintaksi jezika (tzv. „slamanje veba”). Sukob je aktivno trajao do 2003, kada je projekat zvanično napušten [9].

Dve godine kasnije, osnivač Javaskripta Ajk u saradnji za Mozilom počinje da radi na projektu E4X. Pridružuje im se i Makromedija (*Macromedia*; sada Adobi, *Adobe*), u nadi da će standardizovati svoj Ekšnskript 3 u saradnji sa ECMA-om, i time ponovo spojiti razdeljene dijalekte jezika. Ipak, razlika je bila previše velika, što su obe strane počele da shvataju krajem 2007. godine. Otprilike u to vreme, **Daglas Krokford**, koji je tada radio u Jahuu (*Yahoo*), udružuje snage sa Majkrosoftom kao opozicija promenama koje treba da budu definisane standardom **ES4** [50], a svoje stavove iskazuju tako što standard žele da nazovu jednostavno ES3.1, jer unosi samo neznatne promene.

Sve se ovo dešava u vreme kada je Javaskript zajednica pokrenula revoluciju u mogućnostima koje pruža Javaskript, što nesumnjivo otpočinje 2005. godine kada je **Džesi Džejms Geret** objavio članak u kome se prvi put pojavljuje termin Ajaks koji je iskoristio za opis skupa tehnologija izgrađenim nad Javaskriptom. Ovo je bio renesansni period za jezik, i tada počinju da se javljaju biblioteke otvorenog koda kao što su *jQuery*, *Mootools* i *Dojo* koje su do pojave SPA frejmworka bile dominantni alati za izgradnju dinamičkih veb-sajtova i veb-aplikacija.

Iz rata oko konačnog oblika četvrte verzije standarda koji se konačno okončava 2009. godine izlazi Krokford. U Javaskript je posle dvanaest godina uneto svega nekoliko neznatnih promena, a rezultujući standard je preimenovan u **ES5**, kako ne bi došlo do kasnije zabune oko eventualne četvrte verzije o kojoj se već godinama govori. U standard tada dolazi takozvani „strogi režim” (*strict mode*), koji definiše „strožu” verziju jezika, ali na kompatibilan način, kako ne bi narušio već postojeći kod koji se nalazi na vebu.

Dve godine kasnije usledila je mala promena standarda, editorijalnog karaktera, nazvana ES5.1, koja nije bila od većeg značaja za Javaskript.

Godišnji ciklusi

Nove osobine Javaskripta koje nisu ušle u ES5 standard ostale su poznate pod imenom Harmonija (*Harmony*). Tek 2015. godine se ECMA komitet ponovo

sastaje radi definisanja standarda šeste edicije standarda, kada se konačno, posle petanest godina nagomilanih ideja za proširene i poboljšanje jezika, sprovode u delo. Šesta edicija standarda je vrlo kratko pre zvaničnog objavljivanja preimnovana iz ES6 u **ES2015**, zbog ideje da se u nastavku standard za Javaskript obnavlja na svakih godinu dana, kako naredne promene ne bi bile toliko velike kao ova.

ES2015 je drastično promenio izgled Javaskripta, čime je omogućeno znatno jednostavnije pisanje složenih aplikacija [47]. Nesumnjivo najznačajnija novina u jeziku su moduli koji omogućuju podelu koda u fajlove, ali je tu i intuitivnija sintaksa za defisanje klasa, iteratori i generatori (koji će poslužiti kao osnova za `async/await` u ES2017), generatorski izrazi, `for...of` petlje, „streličaste” funkcije za sintaksno jednostavnije pisanje funkcionalnog koda, binarni podaci, tipizirani nizovi, kolekcije (mape, skupovi, slabe mape), promisi, dodatne funkcije na `Math` objektu, bolja refleksija, kao i posrednici za metaprogramiranje virtuelnih objekata i omotača.

Sedma edicija (ES2016) objavljena je, po dogovoru, 2016. godine i uvodi samo blage promene [45]: operator za stepenovanje `**` i `Array#includes`. Sledeća edicija uvodi dosta promena u jezik koje omogućavaju da se procesorska moć iskoristi na znatno višem nivou nego ranije. Među novinama iz **ES2017** izdvaja se mogućnost paralelnog programiranja, izvođenja atomičnih operacija ukoliko ih procesor podržava, nadgledanja tokova podataka (*observable streams*) i definisanja tipova podataka koji omogućuju SIMD programiranje. Sem toga, uvedena je sintaksna integracija promisa i generatora pomoću ključnih reči `async` i `await`, a bolji Simboli predstavljaju korak bliže predefinisanju operatora.

Presek za **ES2018** napravljen je juna 2018. godine, a u novine se ubraja korišćenje *rest/spread* sintakse za objekte pored nizova, asinhrona iteracija (`for await...of` petlje), metoda `finally` nad prototipom promisa i imenovane grupe u regeksima [46].

Odbor TC39

Proces usvajanja nove konstrukcije u jezik nije jednostavan i ne dešava se preko noći. Ekmasript je dizajniran od strane odbora koji se naziva **TC39**, a sačinjen je od kompanija među kojima su i kreatori svih značajnijih brauzera. Redovnim sastancima prisustvuju delegati koje biraju kompanije-članice, a nekada se pridružuju i spoljni eksperti za određene oblasti. Sažeci sastanka su javno dostupni.

Odbor je organizovan uz izdanje ES2016, kada je zvanično potvrđeno da će se jezik nadograđivati na godišnjem nivou. Konačnu odluku o uvođenju osobine u Javaskript donosi većina iz odbora, ali pod uslovom da nijedna članica nije strogo protiv ideje. Kako su mnoge članice kompanije koje rade na razviću brauzera, za njih slaganje sa promenom jezika sa sobom nosi i teret odgovornosti oko njihove implementacije.

Svaki predlog za dopunu jezika prolazi kroz nekoliko *faza zrelosti*, počev od nulte faze [48]. Prelazak u sledeću fazu mora biti odobren od strane odbora. Svaki predlog počinje kao neformalna, tzv. **slamnata** (*strawman*) ideja – ovo je faza 0. Ukoliko TC39 smatra da predlog ima potencijal, on se dodaju na listu slamnatih predloga.

U fazi 1 se za ideju formalno može govoriti kao o **predlogu** (*proposal*). Za predlog se bira jedan *šampion* koji je za njega odgovoran, pri čemu jedan od šampiona i ko-šampiona mora da bude član odbora. U ovoj fazi se daje tekstualni opis ideje, praćen primerima, diskusiji o API-ju, semantici koja se uvodi u jezik i algoritmima koji će se koristiti za obavljanje metoda koje se uvode u jezik. Takođe se identifikuju potencijalne prepreke u usvajanju predloga u specifikaciju jezika. Ukoliko predlog to dozvoljava, neophodno je da se implementiraju polifilovi¹ i nekoliko demonstracija koje ih koriste.

Prva faza samo iskazuje da je odbor zainteresovan za predlog. Tek kada predlog pređe u drugu fazu, počinje da se piše **nacrt** (*draft*) – to je prva verzija onoga što treba da postane deo formalne specifikacije jezika. Nadalje se očekuju samo inkrementalne promene.

Treću fazu čine **kandidati** (*candidate*). U ovoj fazi, specifikacija je kompletirana i moraju da postoje barem dve međusobno nezavisne implementacije u popularnim brauzerima.

Završeni (*finished*) predlozi su spremni da se dodaju u standard. Da bi se predlog našao u ovoj fazi, potrebno je da obe implementacije zadovoljavaju „Test262”².

Biblioteke i frejmvorci

Teško je govoriti o jednostraničnim aplikacijama a ne pomenuti biblioteke i frejmvorke koji su nastali kako bi autorima olakšali njihovo razviće, a definisati koji frejmvork je prvi napravljen konkretno za razviće SPA aplikacija nije nimalo lakši zadatak. Ne samo da je termin „*single-page application*” postao rasprostranjen mnogo kasnije nego što se javili alati koji su se koristili za njih, već se i očekivanja od SPA frejmvorka vremenom menjaju – neke osobine frejmvorka koje se danas smatraju standardom (npr. SSR) u vreme prvih frejmvorka nisu ni bile razmatrane.

Posle Džesi Džejms Garetove objave članka „*Ajax: A New Approach to Web Applications*” iz 2005. godine [27], gde je opisao kako su aplikacije –u to vreme revolucionarne verzije *Gmail*-a i *Google Maps*-a koje su vršile navigacije ne

¹Polifil (*polyfill*) je funkcija koja implementira funkcionalnost jezika koja u trenutnoj verziji ne postoji. Osnanjanje se na postojeće osobine jezika. Obično se implementiraju promenom prototipskih metoda ili dodavanjem vrednosti u globalni objekat kao što je `window`.

²*Test262* je sertifikat kojim se potvrđuje usaglašenost najnovije verzije nacrtane specifikacije i implementacije određenih delova ECMA standarda.

zahtevajući novu stranicu sa servera– razvijene u Guglu [54], termin „Ajaks” se brzo prihvata i javljaju se biblioteke koje omogućavaju jednostavnije korišćenje skupa tehnologija koje čine Ajaks. U ovo vreme koristi se termin „Ajaks frejmwork” za novonastale biblioteke. Među njima se neosporivo kao predvodnik izdvaja *jQuery*, nastao početkom 2006. godine [56]. *jQuery* i danas zadržava mesto najkorišćenije biblioteke. Koristi ga preko devedeset miliona sajtova, pritom preko 80% od milion najposećenijih sajtova [33].

jQuery prate biblioteke sa sličnim ciljem – na primer, *Dojo Toolkit* (2005), *YUI* (2006), *Ext JS* (2006). Najpopularniji konkurent *jQuery*-ju bio je *MooTools* (2006), danas uglavnom poznat po tome što je menjao prototip postojećim Javaskript objektima, što se smatra lošom praksom – i to s razlogom. Naime, jedna od dodatih funkcija bila je metoda `contains` nad `String`, što je kasnije prouzrokovalo probleme kada se ista funkcija pojavila u ES2015 standardu, ali sa drugačijim potpisom. Fajerfoks je u verziji 18 dodao standardu metodu, ali to je prouzrokovalo da dođe do greške na sajtovima koji koriste *Mootools* [17]. Iako je *MooTools* promenio potpis u verziji 1.5, ranije verzije su ostale dovoljno rasprostranjene na internetu, pa je donesena odluka da se `contains` u standardu preimenuje u `includes`.

Miško Heverli je tokom 2009. predstavio svetu na svom blogu projekat pod imenom `<angular/>` [31], uz prateće predavanje sledeće godine pod imenom „`<angular/>`: potpuno drugačiji način za razvije ajaks aplikacija” [30]. Biblioteke kao *jQuery* koristile su se za olakšan rad sa DOM-om. `<angular/>`, kasnije preimnovan u **AngularJS** omogućio je jednostavnu sinhronizaciju „prave” vrednosti podataka (modela) i DOM-a (pogleda). AngularJS aplikacije sastoje se od kontrolera i šablona – kontroleri su odgovorni za manipulaciju stanjem modela, a šabloni za prikaz korisniku.

Ubrzo je u usledilo još frejmworka sa sličnom idejom – među najpopularnijim se izdvajaju *Backbone* (2010), *Knockout* (2010), *Ember* (2011).

Početkom 2010, Marsel Laverdet, inženjer iz Fejsbuka, objavio je članak „XHP: novi način za pisanje PHP-a”, u kome XHP opisuje kao „proširenje PHP-a koje nadograđuje sintaksu jezika kako bi front-end kod bio jednostavniji za razumevanje i kako bi se smanjio rizik od XSS napada” [36]. Sintaksa koju XHP nudi inspirisana je (u međuvremenu odbačenim) Ekmaskript predlogom E4X (*ECMAScript for XML*) – u pitanju je sintaksa koja dozvoljava mešanje XML-a i Ekmaskript koda.

Fejsbuk je već u to vreme koristio frejmwork **React** razvijen za interne potrebe, ali kupovinom Instagrama 2012. dobija dobar razlog da izdvoji kôd frejmworka u posebni paket kako bi Instagram mogao da bude razvijen istom metodom. Na konferenciji *JS ConfUS* 2013, Džordan Vok je predstavio *React* i objavio da će biti otvorenog koda.

Paralelno, Javaskript kao jezik prolazi kroz revoluciju (v. §1.2), pa se javlja i veliki broj alata koji nisu frejmvorci, ali pomažu u pisanju Javaskriptpa. Nodov repozitorijum **npm** postaje vodeći način za deljenje Javaskript biblioteke sa

ostatkom zajednice, a novi Javaskript frejmvorci počinju da se pojavljuju naizgled niotkuda. Neka imena i danas ostaju prepoznatljiva, makar istorijski, ali masa dobija „pet minuta slave” i onda pada u zaborav. Na slici 1.1 prikazani su neki Javaskript frejmvoci.

Angular	AngularJS	Atv.js	Aurelia	Backbone	Batman	CanJS	Cappuccino
Chaplin	Cycle.js	Derby	Deku	Ember.js	Espresso.js	Ext JS	Feathers
GWT	Hyperapp	jsblocks	Keo	Knockout	LiquidLavva	Marko	Marionette
Mercury	Meteor	Mithril	Mojito	Nativescript	Omniscient	Polymer	Preact
Ractive	React	Rivets	Riot	Ripple	Sammy	SnackJS	Spine
Stapes.js	Throax	Vue.js	XAJA	Way	WebRx	...	

Slika 1.1: Neki od Javaskript frejmvorka, sortirani alfabetski.

Problem izbora frejmvorka ne prolazi nezapaženo od strane zajednice; najčešće se pominje pod terminom „zamor od Javaskripta” (*JavaScript fatigue*), navodeći da srž problema leži u tzv. paralizi izbora [41, 38, 39]. Ekipa koja stoji iza projekta *TodoMVC* ovaj fenomen naziva YAFS (*Yet Another Framework Syndrome*, sindrom još jednog frejmvorka) [1].

Ipak, i pored veoma raznovrsnog ekosistema i razlika kojima se svaki frejmvork ponaosob ističe, svi oni dele isti cilj: olakšati način ažuriranja pogleda kada dođe do promene stanja.

Detekcija promena

Stanje (model) čini skup raznih struktura podataka, bilo da su to nizovi, objekti, brojevi, stringovi, lančane liste. Oni se mapiraju u **pogled** u vidu paragrafa, dugmadi, formi, linkova. Proces prevođenja modela u pogled naziva se **renderovanje** i obavlja se manipulacijom DOM-a. Može se shvatiti kao definicija funkcije, ili definicija **mapiranja**, stanja na pogled.

Ovaj zadatak, iako ne uvek trivijalan, sam po sebi nije interesantan; svodi se na proceduru koja na osnovu neke vrste specifikacije pogleda poziva DOM metode, izgrađuje DOM stablo u skladu sa tim i vezuje ga za tekući dokument stranice. Stvari postaju zanimljive kada se postavi pitanje *promene* pogleda.

Promena pogleda nema trivijalno rešenje. Postoji nekoliko glavnih pristupa, a svaki pristup može da ima više varijacija. Detalji izbora algoritma za **detekciju promena**, odnosno algoritma kojim frejmvork garantuje (ili ne) da će pogled uvek biti sinhronizovan sa pogledom, kao i način na koji autor koda obaveštava frejmvork da je došlo do promene – predstavljaju osnovnu metriku na osnovu koje se frejmvorci mogu porediti. Neka od pitanja koja se mogu postaviti su sledeća.

- Koliko se često ažurira pogled?
- Koliko je taj postupak efikasan?
- Šta autor treba da uradi da bi se otpočeo proces ažuriranja?

- Koje preduslove strukture podataka kojima je modelovano stanje moraju da ispune?

Prva generacija Javaskript frejmworka (*Ext JS*, *Dōjō*, *Backbone*) pružila je *mogućnost* da stanje postoji, i *mogućnost* da se pogled ažurira [40]. Ovi frejmvorci pružaju **osnovnu arhitekturu** aplikacije, diktiraju organizaju koda i određuju strukturu datoteka i direktorijuma, ali problem sinhronizacije prepuštaju developeru. Mogu da ponude mehanizam objava i pretplata, ali određivanje toga *šta* treba ažurirati **prepušta se developeru**. Ovako detaljno upravljanje aplikacijom značilo je da su performanse aplikacije najvećim delom zavisile od toga koliko je često i nad kojim delom pogleda je developer pozvao metode za ažuriranje.

Kada aplikacija postane dovoljno velika, ručni pozivi funkcija za ažuriranje nesumnjivo dovode do povećanja kompleksnosti. Svi frejmvorci koji se javljaju posle prve generacije za cilj imaju da reše problem sinhronizacije. Jedan od njih je *Ember.js*, koji omogućuje **definisanje veza** između konkretnog dela stanja i dela pogleda. Slično frejmvorcima prve generacije, kada se stanje promeni, emituje se neki događaj – ali umesto da se prepusti developeru da na taj događaj odreaguje pozivom metode za renderovanje nad nekim delom pogleda, *Ember* ga interno osluškuje i umesto developera automatski poziva odgovarajuću metodu.³

Veoma sličan – a opet na neki način potpuno suprotan – pristup ima *AngularJS*. Rešenje ovog frejmworka takođe se bazira na ideji da se metoda za re-renderovanje treba pozvati samo kada dođe do promene. Međutim, logika je inverzna: kreiraju se „nadgledatelji” (*watcher*) za svaku promenljivu koja je referencirana u šablonu. Kada god se za tim ukaže potreba, frejmwork proverava da li se vrednost promenila od prošlog puta – ako jeste, vrši se ažuriranje DOM-a. Ovaj proces je *AngularJS* nazvao **prljava provera** (*dirty check*), misleći na izmene u modelu kao na „prljave” delove modela koje treba „očistiti” reagovanjem na njih. Glavna mana ovog modela jeste određivanje trenutka *kada* treba pokrenuti takav algoritam. Neophodno je obezbediti da se presretnu⁴ svi pozivi metoda zbog kojih *može* doći do promene u modelu – ovo je bilo koja interakcija sa UI-jem, časovnici i HTTP odgovori. *Angular*, nova verzija frejmworka *AngularJS* napisana iznova, ponaša se konceptualno slično (mada uz potpuno drugačiju implementaciju [57, 42]), ali developer ima mogućnost da određene delove koda optimizuje ukoliko može da garantuje neke preduslove koji postoje u podacima

³Što se *Ember*-a konkretno tiče, „automatski” poziv se postiže tako što se od developera očekuje da ne koristi operator `=` (jednako) za promenu modela, već da koristi posebnu metodu `set`. Ovo nije jedino moguće rešenje – na primer, *Vue.js* koristi sličan način detekcije promene ali se oslanja na getere i setere [8]. Korišćenje *Proxy* objekta (ES2015) je još jedna od alternativa ukoliko nije potrebna podrška za starije brauzere (jer ne postoji način da se simulira konzistentno ponašanje u brauzerima koji nemaju odgovarajuću podršku).

⁴*AngularJS* je u sličnoj situaciji kao *Ember.js* i *Vue.js*. Opisani pristup (sa presretanjem određenih funkcija) je implementacioni detalj i postoji zbog nedostatka Javaskripta u to vreme. Postojao je ES predlog baš za ovaj slučaj korišćenja (`Object.observe`), ali je u potom odbačen u korist predloga *Proxy* kojim se može postići isti efekat. U tom smislu su algoritmi za detekciju promena koje implementiraju *AngularJS*, *Vue.js* i *Ember.js* veoma slični; iako je ugao posmatranja problema malo drugačiji među njima, na kraju se razlika svodi na implementacione detalje i imenovanje faza u ciklusima detekcije.

[52, 43].

Potpuno drugačiji pogled na problem detekcije promena ima *React*. Umesto da traži razlike u *modelu*, *React* traži razlike u *pogledu*. Svaki put kada dođe do promene u modelu, *React* iznova renderuje celo ili deo stabla koji predstavlja pogled – ali, umesto da promene upisuje direktno u DOM (što bi bilo uzuzetno sporo i nepotpuno funkcionalno), pogled se najpre renderuje kao jednostavna struktura podataka u memoriji aplikacije: kao Javaskript objekat. Kada se nova verzija korisničkog interfejsa izrenderuje u ovu strukturu, koja se naziva **virtuelni DOM**, traže se razlike⁵ između stare i nove verzije. U skladu sa rezultatima te razlike, ažurira se samo određeni deo stvarnog DOM-a. *React* zna da je došlo do promene stanja pozivom posebne metode iz klase koju sve komponente moraju da nasleđuju.

Pomoćni alati i novi jezici

Osim frejmworka, razvijaju se i alati koji pomažu autorima prilikom pisanja koda.

Moduli u Javaskriptu

Kao glavni nedostatak Javaskripta se često navodilo to što nema specifikaciju za podelu koda u manje datoteke [29, §10]. Najraniji alati koji su služili da nadomeste ovaj nedostatak su radili jednostavnu konkatenciju [37]. Sa pojavom Noda se javio prvi formalni standard; koristi se funkcija `require` za uvoz datoteka i posebna globalna promenljiva `exports` za izvoz simbola iz njih. Ovaj standard je kasnije dobio ime **CJS** (*CommonJS*). Iako je ovaj standard bio specifičan za Nod i iako nije bio prihvaćen kao Ekmaskript standard, javili su se alati koji dopuštaju developerima da kod namenjen izvršenju u brauzerima (a ne u Nodu) pišu u odvojenim fajlovima i da reference na simbole definišu eksplicitno koristeći `require` i `exports` umesto da se oslanjaju na primitivne metode kao što je konkatencija. Ovi alati su bili prvi **bandleri**. Kao izlaz su davali isključivo jednu `.js` datoteku, tako što su zavisnosti među izvornim datotekama praćene od jedne početne.

Aplikacije su vremenom rasle a SPA metoda za izgradnju aplikacija se koristila sve češće. Postalo je jasno da se dobar deo koda koji se isporučuje korisnicima nikada ne izvrši, jer korisnik nikada ne dođe do određenog dela aplikacije. Da bi se ovaj problem premostio, zajednica je definisala specifikaciju poznatu kao

⁵Tačan način funkcionisanja traženja razlika je implementacioni detalj, ali činjenica da se koristi virtuelni DOM nije. Zapravo, *React* nema šablonsku sintaksu (koja je u slučaju frejmworka kao *Ember.js* i *AngularJS* neophodna kako bi veza sa DOM-om mogla da se uspostavi bez dodatne konfiguracije od strane autora), već se UI definiše pozivima funkcija koje primaju određene parametre u skladu sa reprezentacijom virtuelnih čvorova. Zapravo, *React* je za verziju 16 napisan iznova [10], koristeći novu strategiju traženja razlika u virtuelnom DOM-u, nazvanu *Fiber*, a API je ostao kompatibilan sa prethodnom verzijom.

AMD (*Asynchronous Module Definition*). Ovime je omogućeno da se definišu zavisnosti između datoteka u fazi izvršenja programa na klijentu i da se pribave asihrono, kada se za to javi potreba. Pošto su zavisnosti eksplicitno definisane, dodatne zavisnosti se pribavljaju automatski, pa se s developera skida teret da mora ručno da definiše koje datoteke se prve moraju učitati da bi se učitala druga.

Osim pisanja Javaskripta kroz AMD module, moguće je i pisati ih u odvojenim datotekama a prepustiti nekom devop alatu da ih pretvori u AMD module, bilo da su oni u jednoj datoteci ili u više njih.

I CJS i AMD su dosegli dovoljnu popularnost da se smatraju „standardom” – jedan za Node, a drugi za brauzere. Ali šta je sa modulima za koje ima smisla da istovremeno budu dostupni i u Nodu i u brazeru; na primer, što su biblioteke sa pomoćnim funkcijama kao što je *lodash*?

Kako ne bi postojale dve različite verzije datoteka, osmišljena je nova specifikacija, nazvana **UMD** (*Universal Module Definition*). UMD ne samo da je kompatibilan i sa CJS-om i sa AMD-om, već radi i sa globalno definisanim simbolima.

Paralelno sa ovim, sa porastom popularnosti Javaskripta i veba kao vodeće platforme za razviće aplikacija, počinje da se intenzivnije radi se na specifikaciji Harmonije, nove verzije EkmaSkript specifikacije koja je danas poznata kao ES2015. Ovime je problem rada sa više fajlova formalno rešen.

Međutim, postoje dva razloga zašto većina developera ne koristi ES2015 module u produkciji. Prvi je to što i dalje postoji određen procenat ljudi koji koristi brauzere koji nisu implementirali ovaj standard. Drugi je to što se povećava broj zahteva i aplikacija se usporava.

Osim toga što se zahteva više datoteka (što predstavlja više zahteva ka serveru pa samim tim dužu komunikaciju sa istim), ostaje problem minifikacije i drugih tehnika optimizacije koje ili nisu izvodljive ili nisu u dovoljnoj meri izražene. Zbog ovoga se i dalje koriste bandleri koji na složeni način obarđuju fajlove kako bi pronašli njihove međuzavisnosti i od njih stvorile jedan ili više izlaznih datoteka spremnih za produkciju.

Bandleri i minifikacija

Banlovanje u Javaskriptu je tehnika koja grupiše odvojene datoteke u cilju smanjenja broja HTTP zahteva koje je potrebno otpočeti kako bi se stranica učitala.

Preteča bandlera su „izvršioci zadataka” (*task runners*), među kojima se ističu *gulp* i *grunt*. Rešenje koje oni nude je daleko od idealnog – developer mora da u nekoj vrsti konfiguracionog fajla definiše redosled kojim fajlovi treba da se učitaju, a izvršilac jednostavno konkatenira sve datoteke, kao da su proizvoljni stringovi, u jedan fajl koji je developer uključivao u stranicu.

Kao jedno od rešenja za ovaj problem nastaje **Vebpek** (*webpack*). Kada procesira aplikaciju, on je potpuno svestan Javaskripta kao jezika u kome je napisana. Kod se parsira, a u nekim slučajevima se delovi koda i interpretiraju; na osnovu toga se rekurzivno izgrađuje graf zavisnosti koji sadrži svaki modul koji je potreban aplikaciji. Kao krajnji proizvod daje manji broj paketa (nekad i samo jedan) koje će brauzer učitati. Veoma je konfigurabilan, što omogućuje granularnije podešavanje veličine i izgleda bandlova, kao i lenjo učitavanje tek kada ih korisnička aplikacija zahteva, čime se ubrzava inicijalno učitavanje aplikacije.

Sličan alat je **Rolap** (*Rollup*), mada je nastao iz drugačijih razloga. Naime, Rolap je prvi bandler koji je prihvatio ES2015 module kao prvoklasni način za učitavanje modula, omogućivši pritom tzv. *tree-shaking* [18]. U pitanju je proces kojim se iz modula dovlače samo simboli koji se koriste, umesto celog modula. Ovo znatno smanjuje veličinu koda prilikom korišćenja eksternih modula, pogotovo modula koji pružaju veliki broj pomoćnih funkcija (kao što je *lodash*).

Zbog koncepta klijenta na vebu, Javaskript ima specifične zahteve kada su performanse u pitanju. Kod desktop aplikacija, pažnja se retko obraća na veličinu izvršne datoteke jer nije od važnosti – korisnici su svesni da mora da prođe određeno vreme kada prvi put preuzimaju aplikaciju i kada je instaliraju, i razlika između sto i dvesta megabajta prolazi nezapaženo. Performanse desktop aplikacija mere su vremenu potrebnom za izvršenje zadatka za koje su namenjene ili u brzini odziva tokom navigacije kroz korisnički interfejs.

Međutim, na vebu su stvari drugačije. Pored istih mera performansi koje imaju desktop aplikacije, ključni faktor za performanse koje će korisnik doživeti je i veličina koda. Da stvar bude komplikovanija, Javaskript je jezik koji se interpretira, što znači da se izvorni kod šalje direktno klijentu. Ukoliko bi autor koda ručno probao da ga učini optimalnim za transfer preko mreže, posle svega nekoliko minuta bi bio neodrživ – bilo bi potrebno ukloniti sve nepotrebne znake beline, što znači da bi se kod pisao u jednoj liniji, a sva imena promenljivih imala bi jedno ili dva slova.

Kako bi se rešio ovaj problem, razvijeni su **minifikatori**. Tehnički, u pitanju su kompilatori koda, ali prevode Javaskript nazad u *Javaskript*. Autor može da tokom razvika aplikacije koristi neminifikovanu verziju, što pomaže prilikom debugiranja, a da pred slanja na produkciju pokrene minifikator. Ipak, ovo je mukotrpan posao pa se minifikatori ugrađuju u build-proces, obično u vidu plugina za bandlera kao što su Vebpek i Rolap.

Najpopularniji minifikator za Ekmaskript je **Terser** (*terser*). Sastoji se od parsera, generatora koda, optimizatora, menglera, analizatora opsega, šetača stabla i transformera stabla [4]. Parser služi za kreiranje apstraktnog sintaksnog stabla (AST) na osnovu Javaskripta koda, koji se dalje koristi od strane optimizatora, što je primarni transformer stabla. Njime se obilazi dobijeni AST i vrše se transformacije nad njima na osnovu unapred utvrđenih pravila, sa ciljem sa se veličina stabla smanji. Na primer: čvorovi koji predstavljaju konstantne izrazi kao

što je $1 + 2$ mogu biti sračunati od strane optimizatora i zamenjeni literalom koji predstavlja dobijenu vrednosti; ukoliko se u uslovu naredbe `if` nađe konstanta `true`, cela negativna grana i sama naredba `if` se mogu ukloniti.

Neke optimizacije nisu „sigurne”, odnosno mogu potencijalno da „slome” kod ukoliko se obave. Jedna takva je promena imena promenljivih (ređe) i imena ključeva u svojstvima objekta (češće), koja se naziva menglovanje. Ukoliko se ovi uslovi ispune, prolazak menglera kroz kod može drastično da smanji veličinu koda pukim prepisivanjem imena.

Javaskript kao assembler veba

Jezici koji se kompajliraju u Javaskript postaju popularni zajedno sa pojavom SPA kao obrasca za izradu aplikacija. Prvi široko rasprostranjen takav jezik je **Kofiskript** (*CoffeeScript*) [11]. Nastao je 2009, a najveću popularnost ima od 2011 do 2014, kada polako počinje da pada u zaborav. U SOJS anketi iz 2016, 60% ispitanika je reklo da je „čulo za njega, ali ne želi da ga uči”, dok 25% kaže da su „ga koristili ranije, ali više ne” – samo 6% je bilo zadovoljno jezikom.

Pad Kofiskripta može da se protumači kao pozitivna stvar, kada se uzme u obzir vreme i razlog zbog koga je on nastao. Naime, razlog zbog koga je postao popularan bile su one strukture koje su kasnije došle uz Ekmaskript 2015. Razlika je u tome što Kofiskript nije *standard*, niti se oslanja na njega – radi se o potpuno proizvoljnoj sintaksi, inspirisanoj Rubijem, Pajtonom i Haskelom. Uz dolazak novih standarda, novih alata i novih jezika, Kofiskript nema šta da pruži. Bez načina za definisanje tipova podataka, nema prednost u ranijem otklanjanju grešaka iz koda, a bez novih osobina u odnosu na novi ES standard, nema razloga za korišćenje. Konciznija i kraća sintaksa, umesto prednosti, počinje da bude dodatni korak koji developeri moraju da savladaju pre nego što mogu da počnu sa radom [12].

Ipak, Kofiskript igra značajnu ulogu u razviću samog Javaskripta. Daglas Krokford izjavio je 2012. da je Kofiskript „lep, elegantan i minimalan” i da bi „voleo da je Javaskript više nalik Kofiskriptu”, nazivajući ga „ljupkim malim jezikom” [22]. Brendan Ajk bio je šampion predloga za „debele streličaste funkcije” u Javaskriptu (`=>`), a deo ideje za konciznu sintaksu je dobio upravo od Kofiskripta [23].

Još jedan popularan jezik koji se pojavljuje je **Dart**; u pitanju je Guglov proizvod iz 2011. godine. Kasnije dobija zvaničnu Ekma specifikaciju i poseban komitet, TC52. Slično Kofiskriptu, ima proizvoljnu sintaksu – ovog puta, inspirisanu C-olikim jezicima. Umesto kraćeg zapisa Javaskript konstrukcija, Dart se osvrće na prvoklasnu podršku za obrasce kao što su klase, miksini, apstraktne klase, itd. Pored ovoga, uvodi i sistem tipova – sistem koji naziva „sigurnim”: koristi „kombinaciju statičke provere tipova i provere tokom izvršenja kako bi se postarao da vrednost koja je dodeljena promenljivoj uvek odgovara statičkom tipu promenljive” [13].

Istovremeno, Majkrosoft radi na projektu pod kodnim imenom „Strada”. Među timom od oko 50 ljudi koji su radili na projektu, ističe se ime Anders Hajlzberg – autor Turbo Paskala i Delfija, i glavni arhitekta u timu za C# u Majkrosoftu. Jezik je prvog oktobra 2012. godine objavljen pod nazivom **TajpSkript**, u verziji 0.8. Prva stabilna verzija (1.0) objavljena je 2014. godine.

Vejn

U prethodna dva poglavlja dat je pregled frejmworkā, sa posebnim osvrtom na neke tehnike na osnovu kojih se može detektovati da je došlo do promene u modelu i tu promenu odraziti na pogled; zatim su pomenuti neki alati i jezici koji omogućavaju da se vrši transformacija koda, a na visokom nivou je dat opis načina funkcionisanja minifikatora.

Ideja ovog rada je spojiti ta dva koraka u jedan. Umesto da se kôd frejmworka i kôd autora minifikuje alatom opšte namene, može se napraviti minifikator koji je svestan postojanja frejmworka. Pošto se sužava skup ulaza u minifikator, moguće je detaljnije obraditi obrasce u kodu i bolje optimizovati izvršnu verziju.

Radi demonstriranja ove tehnike, razvijen je frejmwork *Vejn* (*Wane*). Ideja o minifikatoru i frejmworku spojenim u jedno je proširena korak dalje – ako je minifikator svestan koda frejmworka, da li ima potrebe da kod frejmworka uopšte postoji? Može li da se umesto minifikatora koji menja *postojeći* izvršni kod razviti minifikator koji kreira optimizovan kod *od nule*, na osnovu specifikacije zadate od strane autora?

Odgovor na sva pitanja je *da*, a *Vejn* je primer takve implementacije.

Autor koji za razvoj aplikacije koristi *Vejn* piše kod u *Tajpskriptu*. Ova odluka u trenutnoj verziji implementacije nije od velikog značaja, ali se tip podataka potencijalno može iskoristiti u kasnijim fazama razvića frejmworka kako bi se načinile neke inače nemoguće optimizacije (na sličan način na koji to izvodi *Google Closure Compiler*).

U narednim poglavljima će prvo biti reči o *Tajpskriptu*, a potom o samom *Vejnu* – kako iz ugla autora, tako i iz ugla internih odluka i načina rada.

Tajpskript

Snaga Javaskripta je što u njemu možeš da uradiš sve. Slabost je što stvarno hoćeš.

Reg Brejtvejt

Tajpskript (*TypeScript*) je jezik koji podseća na Javaskript i služi kao njegova zamena za pisanje velikih aplikacija. Dodaje opcione tipove i TC39 predloge koji još uvek nisu ušli u standard, ali su barem u fazi 3. Tajpskript se kompajlira (preciznije, „transpajlira”) u čitljiv Javaskript.

Klasifikcija sistema tipova

Skup pravila na osnovu kojih se svojstvo poznato kao **tip podatka** dodeljuje različitim delovima kompjuterskog programa (promenljive, izrazi, funkcije, moduli, itd) naziva se **sistem tipova podataka**. Služe da bi se formalizovali koncepti koje programeri koriste za algebarske vrednosti, strukture podataka i druge komponente programa, kao što su „logička vrednost”, „niz stringova” i „funkcija koja vraća broj”.

Glavna namera postojanja tipova podataka jeste da minimizuje šanse za pojavu bagova u programima i pruži mogućnost za bolje iskusstvo tokom razvoja softvera, ali se može koristiti i kao sredstvo za sprovođenje optimizacije. **Provera tipova** (*type checking*) je proces kojim se utvrđuje da li su ispoštovana pravila i ograničenja koja nameću tipovi podataka u nekom programskom jeziku. Postoji nekoliko osobina na osnovu kojih se može izvršiti podela načina na koje se vrši provera tipova.

Podele

Prvo pitanje koje se nameće jeste *kada* se dešava provera: da li tokom kompilacije ili tokom izvršenja programa. Za jezike koji se interpretiraju (Javaskript, Pajton, Rubi) ne postoji faza kompilacije pa je jedino mesto gde se može vršiti provera tipova – tokom izvršenja. Za ovakve jezike – u kojima se provera tipova sprovodi tokom izvršenja – se kaže da imaju **dinamičku** proveru tipova. Interpretatori za ovakve jezike obično svakom objektu dodeljuju oznaku tipa u kojoj se nalaze informacije o tipu. Ukoliko se provera vrši tokom kompajliranja, radi se o **statičkoj** proveru. Ako program zadovolji uslove koje nameće analizator kod jezika sa statičkom proverom tipova, onda to daje garanciju da program tokom izvršenja neće da naiđe na određeni skup mogućih grešaka u vezi sa tipovima podataka.

Poznajući trenutak izvršenja provere tipova podataka, postavlja se pitanje *kako* se ona sprovodi. Kod jezika sa **jakim** tipovima podataka, nad kodom su dozvoljene samo one operacije koje imaju smisla u skladu sa semantikom jezika i one koje neće dovesti do gubitka informacija. Za jezike koje ne nameću ova ograničenja kaže se da imaju **slabe** tipove podataka. Ne postoji jasna granica između jakih i slabih tipova, pa se ovi termini koriste uglavnom za poređenje jezika (koji ima „jače” a koji „slabije” tipove) ili za određene osobine jezika (za koju se može reći da je „jaka” a za koju „slaba”).

Javaskript nema podršku za statičko definisanje tipova podataka, a uz to je i slabo tipiziran jezik. Promenljiva kojoj je dodeljen string može u sledećem redu da ima referencu na niz brojeva, a zatim da u nju bude upisana logička promenljiva. Kako je Javaskript jezik koji se interpretira, ni nema prilike da se ovo zabeleži kao greška prilikom kompajliranja; taj proces ne postoji. Međutim, čak i tokom izvršenja programa, ovakvo ponašanje koda se neće tretirati kao greška i interpretator će bez problema upisivati bilo koju vrednost u promenljivu.

Ovakvo ponašanje ne samo da može negativno da utiče na performanse (jer optimizator nije u stanju da predvidi koliko memorije je potrebno da se zauzme za promenljivu), već čini kod nečitljivim i podložnim greškama. Glavna prednost Tajpskripta je uvođenje tipova podataka u Javaskript, ali samo prilikom kompajliranja.

Tajpskript ni na koji način ne utiče na fazu interpretiranja i izvršenja koda. Tajpskript dolazi uz kompajler koji postojeći Tajpskript kod kompajlira u Javaskript. Dobijeni Javaskript kod ne sadrži nikakve informacije o tipovima podataka; ne postoje nikakvi uslovi koji će dovesti do greške ukoliko se promenljivoj dodeli drugačiji tip od navedenog. Umesto toga, Tajpskript ima zadatak da tokom kompajliranja utvrdi tipove i da se postara da developera obavesti o nastaloj nekonzistentnosti tako što emituje grešku.

Da bi se ovo postiglo, Tajpskriptov kompajler obavlja statičku analizu koda tokom koje zaključuje tipove i proverava da li su određeni operatori izvodljivi nad njima. Na primer, iako je se Javaskript interpretator ne buni za pokušaj

sabiranja praznog niza i praznog objekta (`[] + {}`), Tajpskript će prijaviti grešku „Operator ‘+’ cannot be applied to types ‘undefined[]’ and ‘{}’”.

Deklaracije

Kako bi bilo moguće da se koriste postojeće biblioteke pisane u Javaskriptu, u Tajpskript je uveden pojam *deklaracije*. Na primer, Tajpskript ne može automatski da bude svestan postojanja eksterne biblioteke učitane kroz `script` tag u head sekciji HTML dokumenta za koju se piše Tajpskript kod. Deklaracija promenljive ili nekog drugog simbola navodi se ili u posebnom fajlu sa ekstenzijom `.d.ts` ili direktno u postojeći fajl, uz prefiks `declare`.

Deklaracije za mnoge postojeće Javaskript biblioteke se mogu naći na GitHub repozitorijumu Majkrosoftovog projekta *DefinitelyTyped* koji održava zajednica. Svi paketi se mogu preuzeti sa npm repozitorijuma. Radi lakšeg pronalaska deklaracionih fajlova koristi se imenski prostor (*namespace*) `@types`, a za ime deklaracionog paketa se koristi isto ime koje je dodeljeno samom paketu za koji se instaliraju deklaracije. Na primer, za paket `jquery` deklaracije se nalaze na `@types/jquery`.

Deklaracije se mogu dostaviti i zajedno sa samim paketom; ovaj način se preferira u situacijama kada je sam paket već napisan u Tajpskriptu. Koristeći opciju `-declaration`, Tajpskriptov kompajler može, pored `.js` fajlova, generisati i `.d.ts` fajlove, a iz `package.json` vrednosti pod ključem `types` može se pročitati putanja do ovog fajla. Ovako Tajpskriptov kompajler zna gde da pronade potrebne deklaracije.

Definisanje tipova

Neki stariji jezici sa statičkim tipovima podataka od developera zahtevaju da se tipovi uvek definišu eksplicitno. Tajpskript je specifičan po tome što su se njegovi stvaraoci postarali da održe barijeru prelaska od Javaskripta na TajkSkript izuzetno niskom. To je postignuto skroz nekoliko osobina.

Uz odgovarajuća podešavanja kompajlera, migracija na Tajpskript se sastoji od samo jednog koraka: promeniti ekstenziju svim `.js` datotekama u `.ts`. Ovo je omogućeno činjenicom da su tipovi podataka **opcionni**. Svaki Javaskript kod može se formalno prevesti u Tajpskript kod promenom ekstenzije, a Tajpskriptov kompajler će u procesu kompajliranja od `.ts` datoteke generisati kod identičan polaznoj `.js` datoteci. Tajpskript je zbog ovoga s namerom zamišljen kao **nadskup** Javaskripta.

Za **eksplicitno** definisanje tipova koristi se postfixna sintaksa koja je popularna među jezicima koji pružaju opcione tipove (npr. *EkšnSkript* i `F#`). Ispred tipa se navode dve tačke (`:`).

```
const foo: string = "Hello"
```

Informacije o tipovima često nije neophodno ručno specificirati jer tipovi u Tajpskriptu mogu biti **implicitni**. Analizator će, gde god je to moguće, na osnovu analize koda zaključiti o kom je tipu reč. Na primer, ukoliko se konstanta inicijalizuje brojnom vrednošću, za konstantu se implicitno zaključuje da ima tip `number`. Slično, ako se iz funkcije koja prima dva stringa vrati vrednost dobijanja njihovom konkatencijom pomoću polimorfnog operatora `+`, povratnoj vrednosti funkcije se implicitno dodeljuje tip `'string'` jer je upravo tog tipa rezultat primene ovog operatora nad dva stringa.

```
const foo = 2103
const concat = (a: string, b: string) => a + b
```

Strukturalni sistem tipova

Tradicionalni objektno-orijentisani jezici kao što su Java i C koriste sistem tipova u kome se podudaranje između tipova podataka vrši na osnovu njihovih eksplicitno navedenih *imena*. Ovakvi sistemi nazivaju se **nominalni**. Na primer, iako su definisane dve strukture istog oblika, njihova međusobna dodela neće biti moguća jer su imena tipova različita.

```
struct Foo { int baz; };
struct Bar { int baz; };

int main () {
    Foo foo;
    Bar bar;
    foo = bar; // error: no viable overloaded '='
}
```

S druge strane su jezici kod kojih je za određivanje pravila u sistemu tipova presudna njihova *struktura*. Oni se nazivaju **strukturalni** sistemi, u koje spada Tajpskript.

```
interface Foo { baz: number }
interface Bar { baz: number }

let foo!: Foo
let bar!: Bar
foo = bar // ok
```

Konfiguracija

Skup fajlova koje će biti obrađene od strane kompajlera naziva se **kontekst kompilacije** (*compilation context*). Mada je sva podešavanja moguće proslediti

direktno kao argumente CLI aplikacije, obično se, radi preglednosti, za ovo koristi konfiguracioni fajl. U pitanju je JSON¹ fajl za koji se očekuje da bude imenovan `tsconfig.json` i da se nalazi u korenom direktorijumu projekta. Ukoliko to nije slučaj, putanja do konfiguracione datoteke se može proslediti kao argument opcije `--project (-p)`.

Za uključivanje i isključivanje datoteka iz projekta koriste se tri polja u konfiguracionoj datoteci: `files`, `include` i `exclude`. Svi primaju niz stringova. Ukoliko se ne navede nijedna od ove tri opcije, biće uključene sve `.ts` datoteke koje se nalaze u korenom direktorijumu i poddirektorijumima, rekurzivno.

Da bi se naznačilo da datoteka ne pripada projektu, potrebno je dodati je u niz `exclude`. Pored putanja do datoteka, `exclude` opcija prepoznaje i imena direktorijuma i glob² šablone.

Slično, `include` opcija se koristi da bi se ograničio opseg projekta. Takođe razume putanje do datoteka, direktorijuma i blobove. Na primer, sve izvorne datoteke se često nalaze u direktorijumu `src`. Opcija `files` je specifičnija od opcije `include` jer prima samo putanje do datoteka i uglavnom se koristi za manje projekte.

Ponašanje sistema tipa podatka i način na koji će Tajpskript generisati izlaznu datoteku ili datoteke definiše se u sklopu objekta `compilerOptions`. Ovaj ključ prima objekat od preko osamdeset podešavanja. U nastavku će biti iznesene samo najznačajnije.

Izbor ES verzije

Osim što uvodi tipove podataka u Javaskript, Tajpskript služi i za izbor standarda koji aplikacija treba da podrži. Kod je moguće pisati ne samo u tekućoj verziji standarda, već i koristiti predloge koji još uvek nisu ušli u standard.

Kako ne podržavaju svi brauzeri sve TC39 predloge, autor treba da odluči po kom standardu treba da bude napisan emitovan kod. Ovo se određuje poljem `target`.

Najstariji podržani standard je `es3`, i ovo je podrazumevana vrednost za `target`. Moguće je izabrati i konkretne standarde `es5`, `es2015`, `es2016`, `es2017` i `es2018`. Ukoliko developer ne želi da se kod adaptira ni za jedan standard već da ostane u izvornom obliku (naravno, bez tipova), onda se koristi vrednost `esnext`; ovo znači da će izlazni kod sadržati Javaskript koji još uvek nije postao standard, pa postoji dobra šansa da kod bude nevalidan u nekim brauzerima.

¹Nadskup JSON specifikacije u kome se, između ostalog, dopuštaju viseći zarezi i komentari, a omeđivanje ključeva znacima navoda je opciono ukoliko ne sadrži karaktere koji bi doveli do dvosmislenosti; zbog ovoga dobija naziv „JSON za ljude”.

²Glob je string kojim se na koncizan i čitljiv način definiše skup datoteka i/ili direktorijuma; koristi specijalne simbole kao `*`, `**` i `?` da bi se zadale određene „komande”. Primeri: `*.txt`, `node_modules/**/*`, `*.tsx?`.

Moduli

Da bi Tajpskript zadovoljio široke potrebe autorā (v. §1.4.1), podržava više načina da generiše module. Opcija `module` prima jednu od sledećih vrednosti: `None`, `CommonJS`, `AMD`, `System`, `UMD`, i `ES2015`.

Ovo polje je usko povezano sa poljem `target`; `ES2015` je moguće koristiti samo ako je `target` podešeno na `es5` ili niže. S druge strane, jedino ako se izabere `AMD` ili `System` je moguće definisati opciju `outFile` kojom se podešava ime fajla u kojem će se naći izlazni kod.

Globalni tipovi

Osim jezičkih konstrukcija, novi ES standardi u jezik uvode i nove globalne promenljive ili nove metode i svojstva nad postojećim prototipovima. Da bi se Tajpskript kompajlirao bez greške i da bi pružio podršku u vidu tipova podataka za sve metode, autor mora da naglasi koje tipove ili koje grupe tipova želi da vidi globalno dostupne, odnosno za šta očekuje da postoji tokom izvršenja koda u okruženju za koje piše softver.

U ovu svrhu se opciji `lib` prosleđuje niz stringova. Pored celih standarda (od `ES5` do `ES2018`), mogu se uključiti i `ESNext` (kojim se pokrivaju deklaracije za predloge koji još uvek nisu postali standard), `WebWorker` (deklaracije za pokretanje koda u okviru veb-vorkera), kao i određene deklaracije iz standarda pojedinačno: na primer, `ES2015.Core` (osnovni skup funkcionalnosti), `ES2015.Promise` (deklaracije za objekte tipa `Promise` iz 2015), `ES2016.Array.Include` (dodata metoda `include` nad prototipom globalnog objekta `Array`), `ES2018.Promise` (deklaracije za novi `Promise` gde je uključena metoda `finally`), itd.

Emitovanje koda

Sem izbora načina na koji se kod parsira i ciljne verzije standarda, moguće je definisati i šta treba da se emituje.

Ukoliko je Tajpskript potrebno koristiti samo da bi se proverili da li su zadovoljeni tipovi podataka, postavlja se fleg `noEmit`; ovime kompilator neće generisati nijedan fajl ali će izvršiti sve provere nad kodom o tome da li bi se kompilacija završila uspešno ili ne. Slično, fleg `emitDeclarationOnly` se koristi da bi se generisale samo `.d.ts` datoteke – izlaz neće biti kod koji je moguće pokrenuti već će biti kod kojim je moguće obećati Tajpskriptu da postoji neki kod. Ovaj način kompilacije se koristi kada se `.js` datoteke generišu od strane nekog eksternog alata kao što je Babel.

Ukoliko se u kodu koriste određene konstrukcije koje nisu dostupne u Ekmaskript verziji koja je navedena u `target` opciji, Tajpskript može emitovati neke pomoćne funkcije kojim premošćava njihov nedostatak, a koje imaju isti ili dovoljno sličan

efekat na kod kao konstrukcija iz novog standarda koju developer koristi u kodu. Na primer, definisanje klase koja nasleđuje neku drugu klasu koristi ključne reči `class` i `extend`, ali ES5 ne podržava nijednu od njih. Iako je klasu lako imitirati funkcijom, nasleđivanje nije tako trivijalno, pa kompilator dodaje pomoćnu funkciju `__extends` koja prima dva argumenta: izvedenu i osnovnu klasu. Slično, ukoliko se koriste generatorske funkcije, umeće se pomoćna funkcija `__generator`, a ukoliko se koriste ključne reči `async` i `await` onda se pored `__generator` umeće i `__awaiter`.

Potencijalni problem leži u tome što se ove funkcije, ukoliko se koriste, definišu u svakom fajlu ponaosob. Nad velikim projektima ovo može da bude puno bespotrebno ponovljenog koda. Zato je uz verziju 1.5 dodat je fleg `noEmitHelpers`. Kada je ova opcija uključena, Tajpskript i dalje generiše *pozive* ovih pomoćnih funkcija, ali ne i njih same.

Međutim, sada je na developeru da definiše ove pomoćne funkcije – Tajpskript podrazumeva da su globalno definisane. Kako developer ne bi morao ručno da ih definiše, a kako ne bi bile ni generisane u svakoj datoteci, u verziji 2.1 dolazi fleg `importHelpers` sa kojim će generisani kod importovati potrebne module iz biblioteke `tslib`. Ova biblioteka je dostupna na npm repozitorijumu i dovoljno je da je developer instalira u projekat kao zavisnost da bi generisani kod radio.

Greške prilikom kompilacije Tajpskript koda se mogu svrstati u dve grupe. Jedna su greške zbog kojih ni Javaskript ne bi bio sintaksno ispravan, i zbog njih se kompilacija uvek obustavlja neuspešno, bez emitovanja datoteka. Drugu grupu čine semantičke greške u vidu nelegalnih operacija u vezi s tipovima podataka – iako je dodela stringa broju nemoguća, ovo je moguće kompajlirati u validan Javaskript. Postavkom flega `noEmitOnError`, ponašanje prilikom nailaska na drugi tip greške se može promeniti, tj. moguće je zabraniti da se emituju `.js` datoteke sve dok ima bilo kakvih grešaka u `.ts` kodu. Ova opcija se obično ostavlja isključena prilikom migracije sa Javaskript projekta na Tajpskript, ali i prilikom developmenta – nekada je korisno da se pokrene kod iako nije u potpunosti ispravan, kako bi se nešto brzo testiralo.

Lintanje koda

Iako za to postoji poseban alat pod nazivom `tslint`, i sam Tajpskript kompajler nudi neka osnovna podešavanja u vezi s lintanjem koda. Na primer, ukoliko se uključi fleg `noFallthroughCasesInSwitch`, kompilator će prijaviti grešku ukoliko naiđe na `case` segment koji se pretapa u sledeći (odnosno ne završava se sa `break`, `return` ili `throw`).

Flegovi `noUnusedLocals` i `noUnusedParameters` služe da bi autor koda bio obavešten o tome da postoje definicije lokalnih promenljivih ili konstanti, odnosno parametara funkcije, a da su one neiskorišćene u ostatku koda. Ovakve pojave obično znače da postoji logička greška u kodu ili da postoji zaostatak iz neke ranije verzije koda.

Strogi režim

Kao što je već rečeno, Tajpskript je nadskup Javaskripta i zamišljen je tako da migracija sa Javaskripta bude izuzetno jednostavna. Međutim, za projekte koji se otpočetak pišu u Tajpskriptu (ili za projekte koji su u potpunosti migrirani), često se javlja potreba za strožim proveravama.

Na primer, u slučaju da kompilator ne može da zaključi o kom je tipu reč, automatski mu se dodeljuje univerzalni tip `any`. Ovakvo ponašanje je korisno prilikom migracije, ali zapravo može dovesti do greške prilikom izvršenja koda ukoliko se autor nije postarao da vodi računa o tipu takvog simbola. Kako bi se ovakve situacije izbegle, koristi se fleg `noImplicitAny`.

Postoji čitav skup opcija sličnih ovoj: `noImplicitAny`, `noImplicitThis`, `always-Strict`, `strictNullChecks`, `strictFunctionTypes` i `strictPropertyInitialization`. Navedena lista je vezana za verziju 3.0, ali se može promeniti u budućim verzijama. Kako autor koji želi da provera tipova bude što stroža ne bi morao da prilikom svakog ažuriranja verzije Tajpskripta vodi računa o tome da istraži novododata podešavanja i uključi nove flegove, postoji omni-opcija `strict` kojom se uključuju svi flegovi koji se tiču strogog režima.

Tipovi

U Javaskriptu, podatak koji nije objekat i nema nijednu metodu naziva se primitivni tip. Ima ih šest: `string`, `number`, `boolean`, `null`, `undefined` i `symbol`. Za svaki od ovih tipova postoji i odgovarajući statički tip u Tajpskriptu, sa istim imenom.

```
const aNumber: number = 2103
const aString: string = 'hello'
const aBoolean: boolean = true
```

Kao kontrast primitivnim tipovima, svi neprimitivni tipovi pripadaju tipu `object` (od verzije 2.2).

Osim primitivnih tipova, Tajpskript nudi i neke ambijentalne tipove, u zavisnosti od izabranih opcija kroz polje `lib` u konfiguracionom fajlu. Ambijentalni tipovi vezani za postojeće globalne Javaskript objekte, što znači da postoje tipovi poput `Object` i `Function`. Na primer, neke od metoda definisane nad tipom `Object` su `toString()`: `string` i `hasOwnProperty(v: string): boolean`. Kada se kaže da je u JavaSkritu sve objekat, misli se na to da se u svakom prototipskom lancu na vrhu nalazi `Object`, što znači da su metode kao `toString` i `hasOwnProperty` dostupne svakoj vrednosti, bilo da se radi o primitivnim ili neprimitivnim tipovima.

Prema tome, `Object` opisuje ono što je zajedničko za svaki objekat u Javaskriptu (uključujući i primitivne vrednosti kao što su brojevi); `object` odgovara užem

skupu jer isključuje primitivne tipove.

Za preciznije definisanje objekata koristi se posebna sintaksa, koja podseća na inicijalizaciju objekata. Unutar para vitičastih zagrada navode se ključevi za koje se očekuje da postoje, a zatim se, posle dvotačke, navodi njihov tip. Ova lista se odvaja zarezima.

```
const anObject: { foo: number, bar: string } = {  
  foo: 2103,  
  bar: 'hello',  
}
```

Počev od verzije 1.4, moguće je definisati alijase za tipove. Ovime je omogućeno da se tipovi lakše koriste kroz program i da se međusobno referenciraju.

```
type MyObject = { foo: number, bar: string }  
const anObject: MyObject = { foo: 2103, bar: 'hello' }
```

Svojstvo u definiciji strukture objekta se može označiti kao opcioni pomoću simbola `?` koji se dodaje ispred dvotačke.

Od verzije 2.0 se ispred imena svojstva može dodati modifikator `readonly`. Ukoliko on postoji, dodela vrednosti tom svojstvu je moguća samo prilikom inicijalizacije.

Mada se za tipiziranje funkcija može koristiti tip `Function`, ovo je retko dovoljno jer ne pruža uvid u to koliko argumenata funkcija ima, kog su oni tipa i koja im je povratna vrednost. Za preciznije definisanje tipa funkcije koristi se posebna sintaksa koja podseća na streličastu funkciju u Javaskriptu. Argumenti funkcije su oblika `arg: Tip`, razdvojeni zarezom i omeđeni oblim zagrada. Desno od ove liste se, nakon debele strelice (`=>`), daje tip povratne vrednosti.

```
type MyFn = (foo: number, bar: string) => boolean  
const myFn: MyFn = (foo, bar) => foo > 21 && bar.length < 3
```

Tipovi argumenata i povratne vrednosti se mogu definisati i prilikom same definicije funkcije.

```
const myFn = (foo: number, bar: string): boolean => {  
  return foo > 21 && bar.length < 3  
}
```

U gornjem primeru bi tip `boolean` kao tip povratne vrednosti mogao i sâm analizator da zaključi na osnovu poznavanja tipova koje vraćaju operatori `&&`, `>` i `<`, pa ga nije neophodno definisati. Prednost definisanja tipova jeste što predstavlja dokumentaciju čitljivu ljudima, pa čovek može brže ustanoviti koji je povratni tip (ne mora da analizira telo funkcije, već je dovoljno da pročita samo potpis). Eksplicitno navođenje tipa povratne vrednosti je korisno i zbog provere tipa povratne vrednosti u samom telu funkcije.

Osim streličastih funkcija, ovaj način se koristi i kod klasične definicije funkcije pomoću ključne reči `function`.

```
function myFn (foo: number, bar: string): boolean {
    return foo > 21 && bar.length < 3
}
```

Prilikom provere da li je funkcija pozvana ispravno, pored tipova argumenata posmatra se i njihov broj. Javaskript nema ograničenje po pitanju broja argumenata sa kojima se funkcija može pozvati; ukoliko postoje argumenti viška, oni se ignorišu, a ukoliko se neki argumenti ne dodele, za njihovu vrednost se uzima `undefined`.

U Tajpskriptu se na broj argumenata može uticati na dva načina. Proizvoljan broj argumenata zdesna se može označiti kao opcioni, dodavanjem simbola `?` posle simboličnog imena argumenta. Svi opcioni argumenti moraju biti uzastopni i jedan od njih mora biti poslednji. Ovime je sprečeno da se obavezan argument nađe posle opcionog. S druge strane, broj argumenata se može povećati na beskonačnost ukoliko se ispred simboličnog imena poslednjeg argumenta doda token `...`.

Na primer, globalno dostupna funkcija `parseInt` kao prvi parametar prima `string` koji treba parsirati, a drugi parametar, koji je opcioni, predstavlja brojnu osnovu.

```
declare function parseInt (s: string, radix?: number): number
```

Metoda `push` definisana nad prototipom globalnog objekta `Array` prima proizvoljan broj argumenata – to su elementi koje treba dodati na kraj niza.

```
interface Array<T> {
    push (...items: T[]): number;
}
```

Unija i presek

Čest slučaj je da se nekoj promenljivoj mogu dodeliti može dodeliti vrednost koja može biti različitog tipa. Ovaj slučaj se u Tajpskriptu iskazuje pomoću **unija**. Mogući tipovi se navode razdvojeni simbolom `|`.

```
const a: number | string = 2103
const b: number | string = 'hello'
```

Kada se pristupa svojstvima sa unije, moguće je pristupiti samo onim koji postoje nad oba tipa. U opštem slučaju je ovo `Object`, ali u zavisnosti od tipova koji se nalaze u uniji ih može biti više. Na primer, i nizovi i stringovi imaju svojstvo `length`.

Interfejsi i klase

Drugi način za definisanje oblika objekata su interfejsi. Definišu se ključnom rečju `interface`, nakon koje se navodi ime a potom lista parova oblika ime:

Tip. Glavna sintakсна razlika liste jeste što se umesto zareza za razdvajanje parova mogu koristiti tačka-zarez ili novi red.

```
interface User {  
    name: string  
    age: number  
}
```

Na sličan način se definišu i klase.

```
class User {  
    name: string  
    age: number  
}
```

Mada se mogu koristiti na sličan način kao interfejsi, suštinska razlika između klasa i interfejsa je to što klasa postoji tokom izvršenja programa, a interfejs je samo tip podatka koji služi Tajpskriptovom kompilatoru za proveru. Interfejs ni na koji način nije odražen u kodu tokom izvršenja, pa je nemoguće koristiti operatore kao `instanceof` da bi se proverilo da li objekat implementira neki interfejs.

Osnovni tipovi

Pored šest tipova izvedenih iz primitivnih tipova koje opisuju Javaskript objekte i jednog tipa za sve neprimitivne vrednosti, Tajpskript definiše još četiri osnovna tipa: `unknown` (od 3.0), `any`, `void` i `never` (od 2.0).

Jedan od glavnih dodataka koji dolaze uz verziju 3.0 jeste **tip `unknown`**. Pomoću `unknown` se opisuju simboli za koje je tip nepoznat, odnosno za simbole za koje se ne može garantovati kakva će se vrednost naći u njima. Tip `unknown` se najčešće koristi za dinamički učitane podatke, kao što su odgovori sa servera. `unknown` je **vršni tip** (*top type*). Vršni tip je tip kojem se može dodeliti bilo koji drugi tip, ali se on ne može dodeliti nijednom drugom tipu.

Tip `any` je univerzalni tip. Koristi se za opis tipa promenljivih za koje tip nije od važnosti. Najčešće se koristi tokom migracije ili kada bi pisanje tipa oduzelo previše vremena, a učinak bi bio previše mali. Na primer, ako za neku biblioteku ne postoje deklaracije, programer može najjednostavnije deklarirati globalnu promenljivu kao `declare const foo: any`. Sa ovime, pokušaji pristupa `foo` neće prouzrokovati Tajpskript grešku; s druge strane, kako Tajpskript ne zna o kom je tipu reč, sve operacije nad njime će biti dozvoljene tokom kompajliranja – Tajpskriptova statička analiza se u ovom slučaju ne može koristiti kao garancija da neće doći do greške tokom izvršenja programa.

Svaka funkcija u Javaskriptu, pod uslovom da se uspešno okonča njeno izvršenje (nema beskonačne petlje i ne dođe do greške pre kraja funkcije), mora da vrati neku vrednost. Povratna vrednost navodi se u istom redu posle ključne reči

return. Ukoliko se return ne navede (ili se kontrolom toka zaobiđe), funkcija se završava kada se izvrši ceo blok koji predstavlja njeno telo. U tom slučaju se implicitno iz funkcije vraća vrednost undefined.

Međutim, Tajpskript ipak uvodi **tip void** koji se koristi za povratne vrednosti funkcije koje ne vraćaju ništa (izostavljanjem ključne reči return). Ovime se naglašava da potrošač ne treba da očekuje nikakvu povratnu vrednost od funkcije ili metode, već da se njenim pozivom dešava neki sporedni efekat. Na ovaj način je moguće razlikovati slučaj kada je povratna vrednost funkcije eksplicitno undefined (na primer, kada se u nizu ne nađe traženi rezulta pozivom Array#find) i kada nju treba zanemariti (na primer, Array#forEach samo implicitno vraća undefined, a sporedni efekat se definiše funkcijom koja se prosleđuje kao argument).

Čuvari tipova

Kako je Tajpskript jezik koji se oslanja na Javaskript, mnoge osobine Tajpskripta su prouzrokovane obrascima i čestim šablonima koje developeri koriste dok pišu JavaScript kod.

Na primer, često se na osnovu nekog svojstva utvrđuje o kom tipu objekta je reč.

```
const pointOnPlane = { x: 1, y: 2 }
const pointInSpace = { x: 9, y: 8, z: 7 }

function getHalfPoint (p) {
  if ('z' in p) return { x: p.x / 2, y: p.y / 2 }
  else return { x: p.x / 2, y: p.y / 2, z: p.z / 2 }
}
```

Međutim, utvrđivanje tipova može da bude jako kompleksno, pa u praksi postaje nemoguće utvrditi o kom je tipu reč – barem ne automatskom statičkom analizom koju Tajpskript sprovodi.

```
interface PlanePoint { x: number, y: number }
interface SpacePoint { x: number, y: number, z: number }

function getHalfPoint (p: PlanePoint | SpacePoint): PlanePoint
  | SpacePoint {
  if ('z' in p) return { x: p.x / 2, y: p.y / 2, z: p.z / 2 }
  else return { x: p.x / 2, y: p.y / 2 }
}
```

U prethodnom primeru, verzija 2.6 Tajpskripta prijavljuje grešku na pretposlednjoj liniji prilikom pristupa p.z, sa greškom „Property ‘z’ does not exist on type ‘PlanePoint’”. Zaista, pošto p može da ima bilo koji od dva navedena tipa, u slučaju da se pristupa p.z nad tipom PlanePoint, dolazi do greške. Kompilator

ne može da zaključi da je uslovom 'z' in p zapravo napravljena razlika između tipova.

Da bi se ovaj čest način pisanja koda u Javaskriptu podržao, Tajpskript u verziji 1.6 dodaje mogućnost da autor definiše funkciju koja vraća true ili false, ali kao tip ima defisan **čuvár tipa** (*type guard*). Čuvari se pišu u formi `x is T`, gde je `x` deklarisan parametar u potpisu funkcije, a `T` je bilo koji tip.

```
function isSpacePoint (p: PlanePoint
                      | SpacePoint): p is SpacePoint {
  return 'z' in p
}
```

Ako se 'z' in p iz funkcije `getHalfPoint` iz primera zameni pozivom funkcije-čuvara `isSpacePoint(p)`, Tajpskript više ne prijavljuje grešku; sada može da zaključi da je promenljiva `p` u pozitivnoj grani tipa `PlanePoint`, a u negativnoj grani ono što ostaje kada se iz `PlanePoint | SpacePoint` odbaci `PlanePoint`, dakle `SpacePoint`. Naravno, ovo se oslanja na to je autor dobro definisao čuvara.

U određenim delovima koda, Tajpskript može implicitno da zaključi da se radi o čuváru i da na taj način poboljša tipove podataka na mestima u kodu gde se dešava grananje. Iz verzije u verziju je ovih tačaka u kodu sve više; najpre samo u uslovima kod `if` konstrukcije nad pozitivnom i negativnom granom i kod ternarnog operatora, zatim u `switch` naredbama, da bi se na kraju došlo do toga da se radi potpuna analiza kontrole toka gde su uključeni i rani izlasci iz funkcija i operatori kao `throw`. Sem toga, vremenom se sve više načina pisanja koda prepoznaje kao čuvar.

Dva osnovna i najstarija načina za definisanje čuvara jesu korišćenje `typeof` i `instanceof` operatora.

```
function doSomething (n: string | number): number {
  if (typeof n == 'string') { /* n is of type string here */ }
  else { /* n is of type number here */ }
}
```

Tipovi *null* i *undefined* se iz tipa mogu odstraniti klasičnim poređenjem sa vrednošću `null`.

```
function inc (n: number | null) {
  return n == null ? 0 : n + 1
}
```

Počev od verzije 2.7, operator `in` se takođe može koristiti kao čuvar, pa se sa ovim dodatkom primer koda za funkciju `getHalfPoint` uspešno kompajlira jer se na osnovu 'z' in p može napraviti jasna razlika između dva tipa iz unije.

Tip never

Uz verziju 2.0 dolazi tip `never`. U pitanju je primitivni tip koji predstavlja tip vrednosti koja se nikad neće dobiti. `never` je pod-tip svakog tipa, a nijedan tip nije pod-tip tipa `never`, osim samog tipa `never`. Ovaj tip se prirodno javlja u delovima koda koji nisu dosegljivi, bilo zbog Tajpskriptove analize tipova podataka ili zbog same prirode Javaskripta.

Na primer, funkcije koje se nikad ne završe imaju kao povratni tip `never`. Ovo se može postići beskonačnom petljom ili bezuslovnim bacanjem izuzetka unutar tela funkcije. Sličan tip za povratnu vrednost funkcije je `void`, ali se on koristi za funkcije čije se izvršenje okonča, ali se iz nje ništa ne vrati eksplicitno (odnosno vrati se implicitni `undefined`).

Drugi čest slučaj gde se ovaj tip javlja jeste prilikom ispitivanja tipa simbola pomoću čuvara (bilo implicitnih ili eksplicitnih).

```
function doSomething (x: number | string): any {  
  if (typeof x == 'string') return 1  
  else if (typeof x == 'number') return 2  
  else { /* x is of type never */ }  
}
```

Preklapanje funkcija

U jezicima sa jakim tipovima podataka se često dozvoljava da funkcije imaju isto ime, a da se na osnovu broja i tipa argumenata određuje koju od njih treba pozvati. Međutim, kako je Javaskript jezik sa izuzetno slabim tipovima podataka, ovaj način preklapanja funkcija nije moguć. Kod koji ispituje broj i topve argumenata mora da definiše autor.

Kako Tajpskript nije u stanju da uvek automatski zaključi o kom je tipu reč, i kako bi bilo potrebno previše menjati izvorni kod i izvoditi zaključke o tome koja je bila developerova namera (što se kosi sa ideologijom Tajpskripta), ovakav kod se i dalje mora pisati u telu funkcije, čak i u Tajpskriptu. Ipak, Tajpskript prepoznaje ovaj šablon i omogućuje da se dobro definišu tipovi ovakvih funkcija; ovo se zove **preklapanje funkcije** (*function overloading*, *function overloads*).

Na primer, iako je u telu funkcije iz primera u prošlom odeljku jasna razlika između dva tipa, povratna vrednost je dvosmislena. Koji god tip da se prosledi u funkciju `getHalfPoint`, iako je on jasno statički definisan, rezultat će biti unija tipova.

```
function getHalfPoint (p: PlanePoint): PlanePoint  
function getHalfPoint (p: SpacePoint): SpacePoint  
function getHalfPoint (p) {  
  if ('z' in p) return { x: p.x / 2, y: p.y / 2, z: p.z / 2 }  
  else return { x: p.x / 2, y: p.y / 2 }
```



```
}
```

Ovako definisana funkcija je preklapljena sa dve deklaracije (prva dva reda). Deklaracija u definiciji funkcije se neće koristiti prilikom poziva funkcije, već samo prilikom provere tipova u samom telu funkcije. Na ovaj način je obezbeđeno da tip povratne vrednosti funkcije bude isti kao tip prosleđenog argumenta.

Generički tipovi

Generički tipovi omogućuju da se ista funkcija koristi sa više različitih tipova, pri čemu je moguće biti eksplicitan o kome se tipu radi. U prethodnom primeru su deklarirana dva potpisa-preklopa kojima se ovo omogućava, ali je istu funkciju moguće zapisati i kraće, korišćenjem generičkih tipova prilikom definicije funkcije.

```
function getHalfPoint<T> (p: T): T { /* ... */ }
```

U gornjem isečku koda, dodata je tipska promenljiva `T` koja omogućava da se uhvati tip koji potpis funkcije prosledi (npr. `SpacePoint`) i da se ta informacija iskoristi u ostatak funkcije (bilo u telu ili u deklaraciji). Ovde se `T` koristi kao tip povratne vrednosti.

Međutim, na ovaj način definisana funkcija će zapravo dati grešku prilikom pokušaja pristupa u negativnoj grani uslova iz tela. Naime, iako se čuvarom obezbeđuje da je tip u pozitivnoj grani `SpacePoint`, u negativnoj grani se zna jedino da *nije* u pitanju `SpacePoint`. Ovo je zato što, kako je funkcija trenutno napisana, `T` može biti bilo šta.

Da bi se ograničio skup tipova koji se može koristiti kao `T`, koristi se format `<T extends U>`, pri čemu je `T` generički tip a `U` tip kojim se određuje kojem skupu tipova mora da pripada tip `T`; drugim rečima, `U` je tip takav da je `T` pod-tip `U`.

```
function getHalfPoint<T extends SpacePoint | PlanePoint>  
  (p: T): T { /* ... */ }
```

Sada ne samo da je tip unutar tela funkcije ispravno definisan, već se i od potražaća zahteva da se za tip unese isključivo `SpacePoint` ili `PlanePoint` (ili tip koji je njihov podskup).

Kada se poziva funkcija sa generičkim argumentima, posle imena se navodi tip.

```
getHalfPoint<SpacePoint>({ x: 9, y: 8, z: 7 })
```

Međutim, `TajpSkipt` omogućava i drugačiji način da se funkcija pozove. Ukoliko je moguće zaključiti stvarni tip generičkog tipa na osnovu stvarnih argumenata prosleđenih funkciji, onda se specificiranje generičkog tipa može izostaviti – kompajler će sam popuniti prazninu.

```
getHalfPoint({ x: 9, y: 8, z: 7 })
```

Osim toga, ako se funkciji prosledi tip `SpacePoint | PlanePoint`, povratna vrednost će takođe biti `SpacePoint | PlanePoint`. Ovo je posledica toga što takva unija zadovoljava `extends` uslov naveden u deklaraciji generičkog argumenta funkcije.

Pored metoda, i interfejsi i klase mogu biti generički. Na primer, `Array` je generički tip, pa se umesto kraćeg zapisa `T[]` može koristiti pun zapis `Array<T>`.

Uslovni tipovi

Veliki pomak u mogućnostima koje pruža Tajpskript načinjen je verzijom 2.8 uz koju dolaze uslovni tipovi. Uslovni tipovi omogućavaju autoru da bude izabran jedan od dva tipa na osnovu ispunjenja uslova postavljenim kao test veze sa `extends`. Imaju oblik `T extends U ? X : Y` – kada se `T` može dodeliti `U`, tada je tip `X` a inače je tip `Y`.

Kada je tip koji se proverava „go”, odnosno nije upleten deo nekog tipa-omotača, tada se za uslovni tip kaže da je **distributivan**. Distribucija se vrši automatski nad tipovima koji su delovi unije. Na primer, za `T` se može proslediti `A | B`, pa se rezultat izraza `(A | B) extends U ? X : Y` dobija kao `(A extends U ? X : Y) | (B extends U ? X : Y)`.

Manipulacija tipovima

Tajpskript dolazi uz neke pomoćne tipove koji nemaju nikakav Javaskript ekvivalent, odnosno ne opisuju postojeće ambijentalne simbole. Oni služe da bi developeru omogućili da transformiše tipove, odnosno da manipuliše njima, kako bi se na osnovu postojećih dobili novi. Zaključno sa verzijom 3.1 ih ima deset.

- Za označavanje da su sva svojstva tipa `T` opcioni, koristi se tip `Partial<T>`.
- Obrnuto, `Required<T>` služi da se označi da su svi property obavezni.
- `ReadOnly<T>` označava sve property kao `readonly`.

Neki generički tipovi definisani su zahvaljujući uslovnim tipovima.

- `Exclude<T, U>` iz tipa `T` odbacuje tipove koji se mogu dodeliti tipu `U`. Na primer, `Exclude<A | B | C, C | D>` vraća tipa `A | B`. Definisan je kao `T extends U ? never : T`.
- Suprotno od `Exclude`, tip `Extract<T, U>` definisan je kao `T extends U ? T : never` i služi da bi se iz `T` izdvojili samo tipove koje je moguće dodeliti tipu `U`.

Primeri tipova iz lib

U ovom odeljku će biti prokomentarisano nekoliko značajnih primera iz `lib.*` deklaracionih datoteka koje Tajpskript koristi za ambijentalne deklaracije.

Metoda `filter` deklarirana je nad prototipom `Array` na sledeći način.

```
interface Array<T> {
    filter<S extends T>(callbackfn: (value: T,
                                   index: number,
                                   array: ReadonlyArray<T>,
                                   ) => value is S,
                       thisArg?: any): S[]

    filter(callbackfn: (value: T,
                       index: number,
                       array: ReadonlyArray<T>,
                       ) => any,
          thisArg?: any): T[]
}
```

Metoda je preklapljen sa dva potpisa; pošto se radi o deklaracionoj `.d.ts` datoteci, nema definicije funkcije pa su oba potpisa vidljiva potrošaču. Prilikom poziva metode, preklapljeni potpisi se obilaze redom koji su navedeni, tj. traži se prvo poklapanje.

Zato, ukoliko se kao `callbackfn` prosledi čuvar koji tip `T` svodi na tip `S`, dobijeni rezultat će biti niz čiji su elementi tipa `S`. Ako se pak radi o običnoj funkciji, tip povratne vrednosti neće biti promenjen u odnosu na početni tip koju ima niz nad kojim se metoda poziva.

Osim deklaracija ambijentalnih simbola koji su dostupni tokom izvršenja programa, u `lib` datotekama se nalaze i neki pomoćni tipovi koji mogu poslužiti developeru da izvede jedan tip iz drugog.

Jedan takav tip je `NonNullable`. U pitanju je generički tip koji od prosleđenog argumenta sklanja `undefined` i `null`.

```
type NonNullable<T> = T extends null | undefined ? never : T;
```

Ukoliko je stvarni tip koji se prosledi umesto `T` neki od pod-tipova tipa `null | undefined` (tj. `null | undefined`, `null`, `undefined` ili `never`), onda se dobija tip `never`. Zaista, ako se od nečega što je obavezno *nullable* ukloni ta osobina, dobija se tip koji nikad ne može da ima vrednost. Ako stvarni tip *nije* neki od pod-tipova `null | undefined`, onda se dobija taj isti tip.

Na primer, za `NonNullable<number | string | null>` se radi o distributivnom uslovnom tipu, pa se rezultujući tip dobija na sledeći način.

```
(number | string | null) extends null | undefined ? never : T
```

```

(number extends null | undefined ? never : T) |
(string extends null | undefined ? never : T) |
(null extends null | undefined ? never : T)
  number | string | never
    number | string

```

Uopštenje ovog pomoćnog tipa je `Exclude`, koji prima dva argumenta `T` i `U` i iz `T` odbacuje `U`.

```
type Exclude<T, U> = T extends U ? never : T;
```

AST Tajpskripta

Bezbojne zelene ideje spavaju
besno.¹

Noam Čomski

Apstraktno sintakšno stablo (AST, *Abstract Syntax Tree*) je struktura podataka koju koriste kompajleri radi reprezentacije strukture programskog koda [21]. Predstavlja izlaz iz faze **parsiranja** u procesu kompilacije programskog jezika.

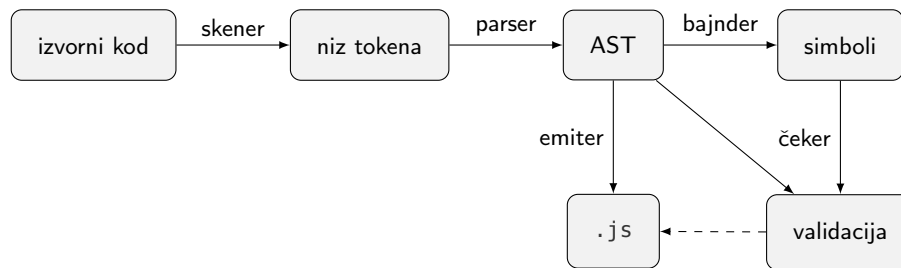
Tajpskriptov kompilator

Tajpskriptov kompajler se u izvornom kodu Tajpskripta [19] nalazi u direktorijumu `src/compiler`. Ključni delovi kompajlera su skener, parser, bajnder, čeker i emiter [55].

Izvorni kod programa pisanog u Tajpskriptu najpre se procesira od strane **skenera** (*scanner*, čitač). Izlaz faze skeniranja je niz skeniranih tokena. Ovako dobijeni tokeni se zatim obrađuju pomoću **parsera** (*parser*, raščlanjivač) i kao izlaz se dobija apstraktno sintakšno stablo. Da bi se iz AST-a dobili simboli, koristi se **bajnder** (*binder*, povezivač). Simboli su osnovni gradivni blok semantike koda – na osnovu njih se, na primer, na osnovu upotrebe simbola prepoznati njegova deklaracija.

Zatim se **čeker** (*checker*, proveravač) koristi kako bi se validirali tipovi podataka. Čeker koristi AST i simbole. **Emiter** (*emitter*, objavljuje) na osnovu AST-a na izlaz daje Javaskript datoteke.

¹Rečenica iz knjige Čomskog pod nazivom „Sintaksičke strukture” iz 1957, kojom je demonstrirao da rečenica može da bude sintakšno ispravna (što znači da se može kreirati njeno lingvističko sintakšno stablo), ali semantički besmislena.



ts-simple-ast

Tajpskript od verzije 1.6 nudi mogućnost za programsko korišćenje API-ja kompilatora [20]. Međutim, ovaj API je još uvek eksperimentalan i prilično nestabilan; u svakoj verziji se javlja veliki broj izmena koje nisu kompatibilne sa prethodnim. Osim toga, veliki broj funkcija je dostupan javno iako nije dokumentovan. Funkcije često zahtevaju da njeni argumenti ili interno stanje kompilatora ispuni neke preduslove koji nisu očigledni iz imena.

Pored ovoga, funkcije su veoma niskog nivoa i često je za neke naizgled jednostavne operacije potrebno pisati veće isečke koda za rekursivni obilazak stabla. Kako bi nadomestio ove nedostatke, **David Šeret** je započeo implementaciju omotača oko Tajpskriptovog AST API-ja kao modul `ts-simple-ast`, koji je zatim sa zajednicom podelio u vidu biblioteke otvorenog koda. Ovaj omotač pruža vrlo jednostavan i intuitivan API za obilazak i modifikaciju AST-a, po cenu neznatnog pada performansi.

Vrste sintakse

Osnovni gradivni blok AST-a je **čvor**; čvorovi implementiraju interfejs `Node`. Osnovni atribut koji karakteriše čvor je **vrsta sintakse** – `SyntaxKind`.

U ovom odeljku iznesene su neke karakteristične sintaksne vrste koje se javljaju u AST-u Tajpskript koda i ključne su za razumevanje poglavlja 5.

Identifier

Identifikatori se često pojavljuju u AST-u kao listovi. Koriste se prilikom deklaracije i definicije, kao i prilikom pristupa. Na primer, definicija funkcije oblika `function foo ()` posle ključne reči `function` (`FunctionKeyword`) ima identifikator (`Identifier`).

Jedna od najznačajnijih metoda kad instancom identifikatora je `getText`. Ova metoda služi za dobavljanje teksta koji se koristi za identifikator, tj. njegovog

imena.

U narednom primeru identifikatori su podvučeni. `this` nije identifikator jer je predstavljen posebnim tokenom `ThisKeyword`.

```
const baz = window.global
function foo (bar) {
  const val = bar + baz + this.toString()
  return val
}
```

PropertyAccessExpression i ElementAccessExpression

U Javaskriptu postoje dva osnovna načina da se **pristupi vrednosti** definisanoj u objektu. Pristup se uvek vrši na osnovu ključa, ali se to sintaksno može zapisati na dva načina. U velikom broju slučajeva koristi se operator `.` (tačka, `DotToken`), ali postoji i alternativni način kojim se pokriva mnogo veći broj slučajeva korišćenja, a to je korišćenje para uglastih zagrada (`[` – `OpenBracketToken`, `]` – `CloseBracketToken`).

Korišćenje tačke predstavljeno je čvorom `PropertyAccessExpression`, a korišćenje zagrada – čvorom `ElementAccessExpression`.

`PropertyAccessExpression` ima metode `getExpression` i `getName`. `getName` se odnosi na desnu stranu i povratna vrednost ove metode je uvek `Identifier` (v. §3.2.1.1). Leva strana dobija se pozivom `getExpression` i može da bude predstavljena različitim tipovima, uključujući i `Identifier` – na primer, u slučaju lanca poziva, može biti ponovo `PropertyAccessExpression`, u opštem slučaju može biti bilo kakav izraz ukoliko se smesti u zagrade (`ParenthesizedExpression`).

S druge strane, `ElementAccessExpression` i sa desne ima mnogo veći spektar potencijalnih čvorova, jer se u paru uglastih zagrada može naći bilo kakav izraz (kome se pristupa metodom `getArgumentExpression`).

CallExpression

Odnosi se na izraze koji predstavljaju **pozive** funkcija i metoda. Prepoznaju se po prisustvu para obliha zagrada unutar kojih se može naći proizvoljan broj argumenata.

`ts-simple-ast` nudi metodu `getExpression` kojom se jednostavno može pribaviti čvor izraza s leve strane zagrada. Najjednostavniji primer je poziv funkcije definisan direktno na osnovu identifikatora i u ovom slučaju je rezultat poziva metode `getExpression` čvor tipa `Identifier`, nad kojim se može pozvati `getText()` za pribavljanje imena čvora. `getExpression` može biti i `PropertyAccessExpression` kada se za pristup funkcije koristi operator `.` (tačka) i

`ElementAccessExpression` kada se koristi `[]` (par uglatih zagrada između kojih se navodi izraz čijom se evaluacijom dobija pristup funkciji). Pored toga, izraz sa leve strane može biti i potpuno proizvoljan `Expression` – na primer, to može biti novi `CallExpression` (za kari-stil izvršenja funkcija) ili pak čitava deklaracija funkcije, koja može biti proizvoljno složena.

```
foo()
this.foo(21)
this[symbol]()
foo()()()
(() => 21)()
```

Tokeni dodele

Mada se dodela najčešće vrši korišćenjem znaka jednakosti (`EqualsToken`), postoji ukupno trinaest tokena kojima se dodela može obaviti. U prvoj koloni sledeće tabele pobrojani su tokeni koji predstavljaju operatore koji se koriste u dodelama, a u drugoj koloni se nalazi po jedan ilustrativni primer za svaki token.

Vrsta sintakse	Primer
<code>EqualsToken</code>	<code>state = 21</code>
<code>PlusEqualsToken</code>	<code>state += 21</code>
<code>MinusEqualsToken</code>	<code>state -= 21</code>
<code>AsteriskEqualsToken</code>	<code>state *= 21</code>
<code>SlashEqualsToken</code>	<code>state /= 21</code>
<code>PercentEqualsToken</code>	<code>state %= 21</code>
<code>AsteriskAsteriskEqualsToken</code>	<code>state **= 21</code>
<code>LessThanLessThanEqualsToken</code>	<code>state <= 21</code>
<code>GreaterThanGreaterThanEqualsToken</code>	<code>state >= 21</code>
<code>GreaterThanGreaterThanGreaterThanEqualsToken</code>	<code>state >>= 21</code>
<code>AmpersandEqualsToken</code>	<code>state &= 21</code>
<code>CaretEqualsToken</code>	<code>state ^= 21</code>
<code>BarEqualsToken</code>	<code>state = 21</code>

Mogućnosti Vejna

wane (of the moon) – have a progressively smaller part of its visible surface illuminated, so that it appears to decrease in size.¹

Google Dictionary

Vejn (*Wane*) je UI frejmwork koji služi za pisanje jednostraničnih veb-aplikacija u Tajpskriptu. Uz frejmwork dolazi CLI i bandler sa optimizatorom, pa se ne javlja potreba za instaliranjem drugih paketa i podešavanjem sistema za kreiranje produkcionog bilda.

Šabloni

S obzirom da se o frejmworku za kreiranje veb-aplikacija, način komunikacije sa korisnicima odvija se pomoću DOM-a. Brauzer kreira DOM na osnovu HTML-a poslatog sa servera, ali i pruža programski pristup istom pomoću Javaskripta. Ideja korišćenja UI frejmworka je da se zaobiđe ovaj proces. Glavni zadatak Vejna je da pogled sa kojim korisnik interaguje održava sinhronizovanim sa modelom, tj. internim stanjem aplikacije. Ovo je omogućeno definisanjem oblika interfejsa kroz *deklarativnu* sintaksu – u pitanju je sintaksa koja podseća na HTML kako bi se developer osećao „kod kuće”, ali se zapravo radi o proizvoljnoj sintaksi.

Termin „deklarativna” se odnosi na to da developer ne upravlja tokom izvršenja koda tokom pisanja šablona. Upravljanje tokom je redosled u kome računar izvršava izraze u skripti [14]. Umesto ovoga, developer samo *deklariše šta* treba da bude prikazano na ekranu korisnika na osnovu stanja modela, a ne i *kako* se ovaj proces odvija. Ovo ne samo da olakšava pisanje šablona, već i omogućuje

¹**opadanje** (meseca) – period tokom kojeg je osvetljena površina meseca postepeno sve manja, dajući privid da mu se veličina smanjuje.

da se vrše interne promene u frejmvorku a da developer toga ne bude svestan – drugim rečima, developer može prostim ažuriranjem verzije frejmvorka koju koristi da unapredi performanse aplikacije.

Šablonska sintaksa je nadskup HTML-a² – to znači da svaki validan HTML predstavlja validan Vejn šablon. Ovo za posledicu ima mogućnost da se postojeći statički sajt vrlo brzo može integrisati u aplikaciju koja se piše u Vejnu, kao i da šablone mogu pisati ljudi koji nemaju iskustva sa pisanjem programa (npr. dizajneri, urednici).

Vezivna sintaksa

Proširenje sintakse HTML-a se ugleda u mogućnosti da se trenutno stanje modela sinhroniše sa pogledom, tj. da se korisnički interfejs osveži svaki put kada dođe do promene modela, u skladu sa pravilima definisanim u šablonima. Ovo je omogućeno vezivnom sintaksom.

Vezivnom sintaksom se mogu definisati izrazi i iskazi. Izrazi su literali i reference na svojstva iz klase, a iskazi su pozivi metoda iz klase.

Literal se definiše pukim navođenjem. Među literale spadaju stringovi ("foo", 'foo'), brojevi (21, 0x3), logičke konstante (true i false), null i undefined. Mogu se referencirati **svojstva** sa klase za koju je šablon vezan ili alijasi koje izlaže direktiva u određenom opsegu, kao i **lanci pristupa** (item, item.name). Ispred svojstva i lanca pristupa se može dodati znak uzvika (!), što znači da će nad podatkom biti primenjen **operator negacije**.

Od iskaza se mogu koristiti jedino **pozivi**. Argumenti se navode u malim zagradama, odvojeni zarezom – kao i u Javaskriptu. Pored izraza, kao argument se može navesti i specijalan simbol # koji označava preuzimanje reference koja se prosleđuje kroz događaj (iz HTML elementa) ili izlaz (iz komponente).

Proizvoljni Javaskript izrazi nisu dozvoljeni u vezivnoj sintaksi – operator negacije predstavlja jedini vid manipulacije koji je moguć nad podacima u okviru definicije šablona. Na primer, za šablon se ne može vezati izraz kao `user && user.name` ili `name.length > 0`. Ukoliko developer želi da u šablonu veže kompleksni izraz koji se računa na osnovu postojećih podataka iz modela, treba da koristi getere na klasi.

Interpolacija

Osnovni vid vezivanja modela jeste ispisivanje stringa direktno na korisnički interfejs. Ovaj vid komunikacije modela i pogleda naziva se interpolacija. Vezivna

²Šabloni predstavljaju isečke korisničkog interfejsa, tj. ne radi se o čitavom HTML dokumentu. Pod „podskupom HTML-a” se misli na podskup sintakse koja je dozvoljena tamo gde se očekuje tzv. *flow content*; v. sekciju 3.2.4.2.2. specifikacije HTML 5.2. U praksi ovi detalji ne predstavljaju prepreku u razumevanju sintakse šablona, pa se koristi termin „podskup HTML-a”.

sintaksa se piše između para dvostrukih vitičastih zagrada.

Na primer, na ekran se može ispisati korisničko ime prijavljenog korisnika.

```
Welcome, {{ username }}!
```

HTML elementi

Kao što je već rečeno, Vejn šabloni su nadskup HTML-a, pa se HTML elementi u šablonima mogu koristiti na standardan način. Ovo uključuje definiciju atributa, prazne elemente (*void elements*) i izostavljanje tagova.

```
<form>
  <label for="name">Name</label>
  <input type="text" id="name" name="name">
</form>
```

Slično parserima HTML-a kod modernih brauzera, parser šablona u Vejnu je otporan na greške – na primer, nezatvoreni tagovi (koji moraju biti zatvoreni po specifikaciji) će se automatski zatvoriti, ali nije definisano tačno u kom trenutku (nailazak na sledeći element, kraj šablona, itd), pa se na ovo ponašanje ni u Vejn šablonima ne treba oslanjati.

Navođenjem atributa u HTML-u se definiše koja će svojstva i koje attribute imati odgovarajući čvor u DOM-u. Svojstva i atributi se koriste da finije odrede semantiku i ponašanje elemenata. Na primer, svojstvo `type` na `<input>` elementu određuje kakav se unos očekuje od korisnika.

Svojstva se mogu sinhronizovati sa modelom tako što se ime svojstva navede u uglastim zagradama. U tom slučaju se vrednost koja se navodi sa desne strane imena tumači kao izraz u vezivnoj sintaksi.

```
<p [id]="myParagraph.uniqueId">...</p>
```

Mapiranje između atributa HTML-a i svojstava DOM čvorova nije uvek jedan-na-jedan. Na primer, mada je `<label for="anId">` validna definicija atributa `for`, ova informacija se sa čvora čita (i postavlja) preko svojstva `htmlFor`. Kako bi se napravila eksplicitna razlika između svojstava i atributa čvorova, za definisanje atributa koristi se prefiks `attr`. Ovo je slično razlici između korišćenja direktnog pristupa svojstvu (`element.svojstvo`) i korišćenju metode `setAttribute` nad čvorom.

```
<label [htmlFor]="anId">...</label>
labelEl.htmlFor = anId
```

```
<label [attr.for]="anId">...</label>
labelEl.setAttribute('for', anId)
```

U slučaju da ime atributa u šablonu sadrži crticu, ono se automatski tumači kao atribut čvora (a ne njegovo svojstvo) jer više ne postoji dvosmislenost – ime svojstava HTML elemenata su izabrana tako da ne sadrže crtice.

```
<input [aria-label]="aLabel">
```

Interpolacija, atributi i svojstva se koriste sa izrazima vezivne sintakse. Koriste se za vezivanje *podataka* za poglede i očuvanje sinhronizacije. Da bi se delalo na korisnikovu interakciju sa interfejsom, potrebno je pretplatiti se na događaje koje emituju HTML elementi. Za ovo se koriste iskazi vezivne sintakse; dakle, definišu se pozivi metoda Drugima rečima, za šablon se vezuje *ponašanje* aplikacije.

Za vezivanje metoda koje treba da budu pozvane na određeni događaj, koristi se sličan zapis kao definisanje svojstava i atributa. Naime, ime događaja se omeđuje parom obliha zagrada, a s desne strane se navodi iskaz kojim se poziva metoda.

```
<button (click)="inc()">Increment</button>
```

Za argumente funkcije se, osim iskaza, može navesti i poseban simbol # – *placeholder*. Njime se može pristupiti objektu koji emituje događaj kako bi se informacije koje on nosi sa sobom mogle iskoristiti u telu metode.

Komponente

Kao i kod svih UI frejmworka, glavna gradivna jedinica u Veju je *komponenta*. Komponente predstavljaju izolovane logičke celine koje se mogu smestiti u korisnički interfejs.

Komponentu treba shvatiti kao prirodnu nadogradnju HTML postojećih HTML elemenata. Na primer, HTML element `HTMLInputElement` ima svoje **ime** na osnovu kojeg se prepoznaje u HTML kodu (`string input`), ima neka **svojstva** na osnovu kojih može da se kontrolišu finese njegovog ponašanja i izgleda koja se mogu specificirati navođenjem HTML atributa ili direktnom manipulacijom kad odgovarajućim `propertijama` Javaskript objekta (npr. `type`), i emituje neke **događaje** uz informaciju o tim događajima na koje se potrošači mogu pretplatiti i reagovati na njih.

Analogno tome, komponente dobijaju **ime** registrovanjem za druge komponente, **ulazi** (*inputs*) koji su sačinjeni od skupa svojstava klase koje je autor komponente proglasio kao delom njenog javnog API-ja, dok su **izlazi** događaji na koje se druge komponente mogu pretplatiti da bi se obavljala komunikacija između njih.

Deklaracija

Komponente se u Veju definišu klasom nad kojom je primenjen dekorator `@Template`, uvezen iz ulazne tačke modula `wane`. Prvi i jedini argument dekoratora je `string` literal kojim se definiše struktura korisničkog interfejsa komponente.

Za stanje komponente koristi se telo klase, gde se standardno mogu definisati njena svojstva i metode. Sva svojstva i metode su dostupne za referenciranje u šablonu.

Pristupna komponenta

Kao što se HTML-om definiše stablo elemenata, u Vejnu se od komponenti pravi *stablo komponenti*. U korenu takvog stabla leži posebna komponentna koja se naziva **pristupna komponenta** (*entry component*).

Pristupnu komponentu treba smestiti u datoteku `index.ts` u folderu `src` i načiniti je podrazumevanim izvozom iz modula – ovako će je Vejn prepoznati kao ulaznu tačku u aplikaciju. Kada se aplikacija pokrene, renderuje se ova komponenta, a ona za sobom povlači sve druge komponente kroz svoj šablon.

Korišćenje

Za razliku od HTML elemenata, koji su uvek globalno dostupni, komponente je potrebno **registrovati** za komponentu kako bi moglo da joj se pristupi iz šablona. Registracija se vrši navođenjem reference na odgovarajuću klasu među argumentima dekoratora `@Register` koji se navodi uz klasu. Ime simbola pod kojim se klasa registruje mora imati veliko početno slovo (tzv. *PascalCase*) kako bi se obezbedilo da neće doći do preklapanja sa imenima postojećih HTML elemenata i potencijalnih veb komponenti koje su registrovane na istoj stranici.

Osim razlike u malim i velikim slovima, postoji još jedna bitna razlika kod korišćenja komponenti u šablonima: simbolička imena za istu klasu (pa samim tim istu komponentu) mogu da se razlikuju među upotrebama u različitim komponentatama. Drugim rečima, ime koje developer da simbolu prilikom registracije je ono pomoću koga komponenta može da se deklarise u šablonu. Na primer, klasa se može preimenovati alijasovanim importom; pored toga, klasa uopšte ne mora da ima ime ukoliko je iz modula izvezena kao podrazumevani simbol, već joj se simboličko ime dodeljuje tek prilikom uvoza.

Registrovana komponenta se u šablon može postaviti kao bilo koji drugi HTML element. Pošto ne može da ima decu, može se koristiti i samo-zatvarajući tag.

```
<Component></Component>  
<Component/>
```

Ulazi

Da bi komponentu bilo moguće parametrizovati, uvodi se pojam **ulaza** (*input*) u komponentu. Ovaj pojam je pandam svojstvima koje imaju HTML elementi.

Ulazi u komponentu se definišu deklarisanjem javnog svojstva na klasi.

Pošto se za proveru ispravnosti tipova u izvornom kodu koristi strogi režim Tajpskripta, ulazima za koje se očekuje da uvek dobiju vrednost iz šablona u kome se koriste dodaje se **!** – stoga, ovo označava **obavezni ulaz**, pa Vejn prijavljuje grešku ukoliko mu se vrednost ne prosledi. Za bi se definisala podrazumevana vrednost ulaza, dovoljno je da se svojstvo jednostavno inicijalizuje. Ulazi sa podrazumevanom vrednošću ne mogu biti obavezni.

```
@Template('...')
class ExampleComponent {
  public input1!: number
  public input2 = 2
  input3: any
  private prop1: any
}
```

U prethodnom primeru definisani su, redom: eksplicitno definisan obavezni ulaz; eksplicitno definisan opcioni ulaz sa podrazumevanom vrednošću 2; implicitno definisan ulaz; obično svojstvo na klasi koje se može iskoristiti za čuvanje stanja komponente.

U šablonima, ulazi se koriste kao da je u pitanju bilo koji HTML atribut, s tim što se ime ulaza omeđuje parom uglastih zagrada, a umesto konkretne vrednosti se očekuje **vezivna sintaksa**.

Osim vezivanja za svojstva klase, veza se može ostvariti i sa alijasima, ukoliko se u šablonu koriste direktive koje ih stvaraju. Više reči o tome u odeljku *o w: for* direktivi.

Izlazi

HTML elementi imaju događaje, a Vejn komponente imaju **izlaze** (*output*). Izlaz se deklarise kao metoda bez tela u telu klase. Da bi se izlaz komponente osluškivao, dodaje se HTML atribut, pri čemu je ime izlaza omeđeno parom oblikih zagrada, a umesto prosleđivanja konkretne vrednosti se očekuje **poziv funkcije**, tj. metode definisane u klasi.

Da bi se pribavila referenca na vrednosti koje se emituju kroz izlaz, u šablonu se koristi specijalni znak **#** kao argument funkcije. Pored njega, argumenti funkcije se mogu referencirati na svojstva klase ili mogu biti literali.

Direktive

Direktive su posebne naredbe koje se umeću u šablone u vidu HTML tagova. Primaju proizvoljan broj čvorova za decu i služe za parametrizaciju načina na

koji će se taj sadržaj prikazati u rezultujućem DOM-u.

Sintaksno, direktive se od HTML elemenata i Vejn komponenti razlikuju po prefiksu `w:`. Zbog svoje prirode, parametri se – umesto navođenjem atributa u vidu parov ključ-vrednost – definišu potpuno proizvoljnom sintaksom koja se piše u nastavku imena, a u sklopu otvarajućeg taga.

Developer nema mogućnost da definiše proizvoljne direktive; sve direktive su sastavni deo Vajna.

Uslovi

Prisustvo ili odsustvo dela šablona može se kontrolisati korišćenjem direktive `w:if`. Uslov na osnovu koga se određuje da li će se grupa elemenata naći u DOM-u ili ne se navodi u vidu izraza vezivne sintakse.

```
<w:if isLoggedIn>
  Welcome, {{ user.name }}!
  <img [src]="user.image" [alt]="user.name"/>
</w:if>
```

U prethodnom primeru se, u zavisnosti od trenutne vrednosti dodeljene svojsvu klase `isLoggedIn`, isečak šablona unutar direktive `w:if` ili prikazuje ili ne.

Nizovi

Predstavnik struktura podataka za rad sa kolekcijama istorodnih elemenata u Javaskriptu su nizovi, implementirani pomoću prototipa `Array`. U šablonima se može vršiti iteracija nad nizovima korišćenjem direktive `w:for`.

U nastavku se navodi sintaksa oblika `(item, index) of items; key: id`, gde je

- `items` – svojstvo ili lanac pristupa sa klase,
- `item` – simboličko ime za jedan element iz niza `items`,
- `index` – simboličko ime za indeks elementa `item` iz niza `items`,
- `id` – lanac pristupa kojim se definiše način na koji se može preuzeti „ključ” po kome se semantički razlikuju elementi niza,

dok su `of`, `;` i `key:` ključne reči kojima se poboljšava čitljivost sintakse.

Simboličko ime za indeks elementa nije obavezno navesti; u slučaju da se ono ne navede, zagrade koje omeđuju simboličko ime za vrednost elementa niza nisu obavezne.

Deo šablona koji je naveden unutar direktive čini šablon koji će se koristiti za kreiranje DOM-a. Ovaj deo šablona parametrizovan je vrednošću elementa koji je

trenutno posećen tokom iteracije (`item`) i, opciono, njegovom brojnom indeksu (`index`). U direktivi, ova imena postaju deo opsega koji se može koristiti za vezivnu sintaksu u šablonima (pored svojstava i metoda klase).

```
<ol>
  <w:for user of users; key: id>
    <li>{{ user.name }} ({{ user.score }})</li>
  </w:for>
</ol>
```

Skraćeni oblik

Vrlo je čest slučaj da se unutar direktive nađe samo jedan HTML element ili samo jedna komponenta. U tim slučajevima, način zapisa direktive se može skratiti tako što se direktiva navodi kao HTML atribut elementa ili komponente, pri čemu se za ključ koristi ime direktive (sa prefiksom `w:`), a kao vrednost se koristi sintaksa koju ta direktiva razume.

```
<span [w:if]="!isLoggedIn">Welcome, guest!</span>
```

```
<ol>
  <li [w:for]="user of users; key: id">
    {{ user.name }} ({{ user.score }})
  </li>
</ol>
```

Dve direktive se ne mogu istovremeno naći na jednom elementu ili jednoj komponenti. Ukoliko se javi potreba za ovakvom strukturom, barem jedna od direktiva se mora zapisati u regularnoj (dužoj) sintaksi.

Stilovi

Kada se govori o korisničkim interfejsima, stilovi su nezaobilazna tema. U brauzerima se oni definišu pomoću CSS-a.

Enkapsulacija

Tradicionalno, CSS se primenjuje globalno, nad celim dokumentom. Kako je osnovna jedinica izgradnje korisničkog interfejsa komponenta, globalni stilovi nisu idealno rešenje jer se onda mora voditi računa o tome da stil jedne komponente ne utiče na stil druge, što remeti mentalni model kojim se komponenta proglašava kao nezavisna celina.

Ovo nije novi problem – uviđen je početkom ere jednostraničnih aplikacija – i počelo se sa radom na njegovom zvaničnom rešenju koje treba da postane deo standarda. Rešenje je nazvano „DOM u senci” (*Shadow DOM*). Specifikacija „opisuje metodu kombinovanja više DOM stabala u jednu hijerarhiju i način na koji ova stabla međusobno interaguju u okviru dokumenta, omogućujući na taj način bolju kompoziciju DOM-a [32]”. Deo specifikacije koji se tiče stilova a povezan je sa DOM-om u senci naziva se *CSS Scoping Module*. Specifikacija još uvek nije finalizirana, ali moderni brauzeri nude eksperimentalnu podršku.

Pozajmljujući ovu ideju, ali ne oslanjajući se na DOM u senci zbog eksperimentalne podrške u brauzerima, stilovi definisani u okviru Vejn komponente će biti vezani isključivo za nju – ovaj pojam se naziva **enkapsulacija stilova**. Stilovi su enkapsulirani i tiču se isključivo dela DOM-a za koji su vezani.

Na primer, ukoliko se u jednoj komponenti promeni boja elementima koji odgovaraju selektoru `button`, a u drugoj se promeni veličina teksta koristeći isti selektor, stilovi će biti primenjeni samo u šablonima komponenti za koje su stilovi definisani – drugim rečima, stilovi neće biti pomešani i nijedno dugme na stranici, bilo unutar tih komponenti ili izvan, neće dobiti mešavinu ova dva stila.

Stilovi se definišu u okviru string literala koji se prosleđuje kao prvi i jedini argument dekoratoru `@Style` koji treba primeniti nad klasom koja opisuje komponentu kojoj se vezuju stilovi.

```
@Template('<button>Red</button>')
@Style{'button { color: red; }'}
class RedButton { }

@Template('<button>Large</button>')
@Style{'button { font-size: 2em; }'}
class LargeButton { }
```

Nadogradnja

Standardni način definisanja CSS-a je proširen posebnim selektorom `:host`. Ime i semantika selektora je pozajmljena iz [34, §3.2], ali je način funkcionisanja emuliran (kao i sama enkapsulacija stilova).

CSS se odnosi na *sadržaj* komponente – bez samog elementa koji predstavlja komponentu, u koji je smešten njen sadržaj. Jedini način da se ovom elementu pristupilo „iznutra” (iz koda same komponente) je pomoću selektora `:host`.

```
:host {
  background-color: red;
}
```

Osim ovoga, podržano je korišćenje registrovanih imena komponenti za selektore (iako se takvo ime – sa velikim početnim slovom – neće odraziti u DOM-u).

```
@Register(Child1, Child2)
@Template('<Child1/> <Child2/>')
@Style('Child1 { color: red; }
      Child2 { font-size: 2em; }')
class Example { }
```

SCSS

Danas se korišćenje nekog alata koji olakšava pisanje CSS-a podrazumeva. Najpopularniji su preprocesori, a kao vodeći se izdvaja **SCSS**, koji se koristi više od pisanja čistog CSS-a [51].

CSS preprocesor je program koji generiše CSS na osnovu sintakse preprocesora. Pojavilo se mnogo ovakvih alata, i većina dodaje novine koje ne postoje u CSS-u: miksin, ugnježdavanje pravila, nasleđivanje, itd. Imaju za cilj da definicije stilova učine čitljivijim i jednostavnijim za održavanje [15].

Stilovi u Vejnu se, umesto u CSS-u, pišu u SCSS-u. Ova opcija se ne može isključiti. Pošto je SCSS striktni nadskup CSS-a, ukoliko developer ne želi da koristi SCSS – jednostavno ne treba da iskoristi nijedan njegov deo i kod će biti potpuno kompatibilan sa običnim CSS-om.

Pokretanje

Pored frejmworka, deo Vejna je CLI koji je primarni način za generisanje izvršne verzije koda, spremne za produkciju. Paket je objavljen tako da se iz komandne linije može koristiti izvršni modul pod imenom wane.

Produkcija

Jedna od dve dostupne komande je komanda `build`.

```
$ yarn wane build
```

Ova komanda pokreće kompilator. Ukoliko aplikacija prođe proces kompilacije uspešno, na standardni izlaz se ispisuju informacije o veličini izlaznih datoteka, a u `dist` folderu se pojavljuju datoteke spremne da bude prebačene na server.

Proces kompilacije izvornog koda se može kontrolisati pomoću konfiguracione datoteke. Ova datoteka treba biti smeštena u korenom direktorijumu Vejn projekta, sačuvana pod imenom `wane.toml`. Sadržaj datoteke treba da bude validna TOML sintaksa³, pri čemu su dozvoljena sledeća dva polja.

³TOML (*Tom's Obvious Minimal Language*) je jednostavan format namenjen za konfiguracione datoteke koji je lako čitljiv ljudima zbog pregledne sintakse i očigledne semantike. Nastao kao kontrast JSON-u koji je namenjen mašinama i YAML-u koji je često previše kompleksno rešenje [44].

- `output.build` je relativna putanja do direktorijuma u kojem treba da budu smeštene izlazne datoteke. Ukoliko direktorijum ne postoji, biće napravljen. Ukoliko postoji, njegov sadržaj će biti obrisano. Ako vrednost nije podešena, podrazumeva se `dist`.
- `debug.pretty` je fleg kojim se kontroliše da li izlazne datoteke treba da prođu proces minifikacije. Na šta se tačno odnosi „minifikacija” nije definisano i nije deo javnog API-ja. Ovime je omogućeno da se ovaj proces unapređuje u narednim verzijama Verna bez potrebe da developer čini promene u kodu. Služi za debugiranje izlaznog koda. Ako vrednost nije podešena, podrazumeva se `false`.

Razviće

Tokom razlika aplikacije može se koristiti posebna komanda `start`. Ovime će biti automatski kreiran dev-server, a `src` direktorijum u kome se nalazi izvorni kod će se osluškivati za promene. Svaki put kada se promena detektuje, proces kompilacije će biti pokrenut iznova, a stranica u brauzeru će biti osvežena kako bi developer mogao da vidi izmene koje su načinjene.

Primer aplikacije

Kompilacija Vejn aplikacije

Dopusti kompilatoru da obavi
jednostavne optimizacije.

Kernigan i Plager, „Elementi stila u
programiranju”

Jedinstvenost Vejna ogleda se u tome što kôd frejmworka, u tradicionalnom smislu – ne postoji. Mada iz ugla developera izgleda da koristi kôd frejmworka tokom izvršenja, preciznije je reći da je Vejn **kompilator**.

Kod napisan od strane developera se koristi da bi se statičkom analizom utvrdile zavisnosti koje postoje između modela i šablona komponente, kao i između komponenti međusobno. Na osnovu ovoga se generiše kod aplikacije koji se koristi tokom izvršenja.

Gledano na najvišem nivou, kompilator čine dva dela: **analizator** izvornog koda i **generator** rezultujućeg koda.

Proces analize koda

Analiza koda počinje od podrazumevanog izvezenog simbola datoteke na putanji `src/index.ts`. Ukoliko se ne radi o klasi, ukoliko klasa nije obeležena dekoratorom `@Template` ili ukoliko je prosleđen šablon nevalidan, faza analize se prekida.

Stablo analizatora fabrika

Prvi korak u analizi je konstrukcija stabla analizatora fabrika komponenti i direktiva. **Fabrika** (*factory*) je funkcija koju generator treba da ispiše; ona treba da se koristi tokom izvršenja aplikacije kako bi se kreirala instanca komponente

ili direktive. U fazi kompilacije, svaka fabrika je predstavljena svojim **analizatorom** (*factory analyzer*). Analizatori sagledavaju kod komponente, poziciju komponentu u stablu, njen API i druge osobine kako bi generator koda mogao da iskoristi ove informacije, na osnovu kojih će biti u mogućnosti da generiše kod u skladu sa namerom developera.

U ovoj fazi analize je važno uspostaviti međusobni odnos komponentata i direktiva, ali i načina na koji su one povezane putem šablona. Na primer, nije dovoljno znati da komponenta A ima za dete komponentu B; važno je dobiti informaciju o tome gde se u *pogledu* komponente A nalazi mesto za koje je vezana komponenta B.

Pogled (*view*) komponente ili direktive je deo šablona kojim ona rukovodi.

Mada struktura stabla koje čini pogled komponente može biti ekvivalentna strukturi samog šablona komponente, ovo nije slučaj ukoliko se u šablonu komponente nalaze direktive. U tom slučaju se pogled komponente završava na mestu gde počinje direktiva, a deo šablona koji se nalazi u unutrašnjosti direktive počinje obradu kao novi šablon. Pored moguće razlike u strukturi, pogled nosi sa sobom mnogo više informacija nego šablon. Može se reći da je šablon struktura koju developer definiše kako bi „objasnio” kompilatoru kako da kreira pogled.

Komponente se prepoznaju po velikom početnom slovu, direktive po prefiksu *w:*. Za ostalo se podrazumeva da se radi o HTML elementima, što znači da su pored postojećih HTML elemenata podržane i veb-komponente. Nailazak na HTML elemente se samo evidentira kao deo pogleda komponente ili direktive koja se trenutno obrađuje. S druge strane, kada se naiđe na HTML tag koji se prepozna kao komponenta ili direktiva, započinje se rekurzivni proces izgradnje stabla pogleda za novu komponentu ili direktivu. Po okončenju procesa, dobijeni analizator se beleži kao dete u analizatoru nad kojim se trenutno vrši obrada.

Sem komponenti i direktiva, **tekst** i **interpolacija** su takođe vrste čvorova koje se nalaze u stablu pogleda.

```
@Template('<div id="a"></div>')
class A { }
```

```
@Template('<div id="b"></div>')
class B { }
```

```
@Register(A, B)
@Template('
  <div id="c"></div>
  <w:if cond>
    <A/>
  </w:if>
  <B/>
')
export default class C { }
```

U primeru TODO, ulazna komponenta je C; od nje počinje izgradnja stabla. Njenom pogledu pripadaju `div#c`, uslovna direktiva `w:if:cond` kao struktura bez unutrašnjeg sadržaja i komponenta B bez unutrašnjeg sadržaja. Struktura koja opsuje decu analizatora fabrike komponente C definiše mapiranje čvora `w:if:cond` na analizator fabrike uslovne direktive, a čvora B na analizator fabrike komponente B, koji ima pogled iste strukture kao i šablon.

Pogled analizatora fabrike uslovne direktive `w:if:cond` ima samo jedan čvor, i to komponentu A. Slično komponenti B iz šablona komponente C, ovaj čvor pokazuje na analizator fabrike komponente A.

Tokom sagledavanja šablona radi generisanja pogleda, za svaki čvor se beleži u kojoj je fabrici **definisan** (*definition factory*) i koja je fabrika **odgovorna** za njega (*responsible factory*). Fabrika u kojoj je čvor definisan je ona fabrika u čijem se *šablonu* nalazi čvor. Odgovorna fabrika je fabrika u čijem se *pogledu* čvor nalazi.

Čvorovi stabla koji sačinjavaju pogled nose sa sobom informaciju o svom tipu i o **povezima** (*binding*) koji su nad njim definisani. Povezi ukazuju na način na koji je postignuta veza između modela i šablona. Na primer, `<div id="a" [className]="b">` je definicija čvora koji predstavlja HTML element sa dva poveza. Prvi povez se odnosi na svojstvo elementa pod nazivom `id` i povezan je sa *literalom* `a` koji je tipa string. Ova informacija je implicitno zaključena i čvor pogleda bi izgledao isto da je umesto `id="a"` autor naveo `[id]='a'`. Drugi povez odnosi se na svojstvo `className` i povezan je sa *svojstvom* `b`.

Strukture koje opisuju *vrednosti* u povezima (literale, svojstva, metode, itd), odnosno „stranu modela” (sparam „strane šablona” koja je opisana čvorovima) nazivaju se **povezane vrednosti** (*bound values*). One igraju značajnu ulogu u fazi generisanja koda u kojoj se određuje *opseg* iz kog se pribavljaju podaci iz modela.

Proces generisanja stabla analizatora fabrika je jedini proces koji se implicitno pokreće u fazi analize. Analize u okviru struktura koje su kreirane ovim procesom su „lenje”, tj. biće pokrenute samo ako informaciju o tome zatraži generator ili sam proces generisanja stabla analizatora fabrika. Rezultati analize se keširaju pa ne postoji problem sa višestrukim pozivom iste funkcije, ma koliko ona bila komplikovana.

Analiza toka koda

Većina informacija koju analizatori fabrika mogu da pruže baziraju se na rezultatima analize toka koda. **Analiza toka koda** je tehnika statičke analize kojom se određuje zamišljena putanja kojom se prolazi prilikom izvršenja koda. Jedan od načina za analizu toka koda, koji se ovde koristi, naziva se **apstraktna interpretacija**. Ovom tehnikom se kod izvršava parcijalno – samo onoliko koliko je dovoljno da bi se utvrdila osobina koja je od interesa.

U ovom odeljku biće definisani neki značajni pomoćni algoritmi koji se koriste u analizi koda komponenti.

Algoritmi se oslanjaju na obilazak i manipulaciju apstraktnog sintaksnog stabla (AST, *abstract syntax tree*) koda napisanog u Tajpskriptu. Mada se sintaksa programa predstavlja strukturom stabla, tok izvršenja se predstavlja proizvoljnim grafom. Na primer, metoda *A* može pozivati metodu *B*, a metoda *B* metodu *A*; metoda može pozivati i samu sebe. Zato se za obilazak toka koriste algoritmi za obilazak grafova. Pošto su veze između čvorova neoznačene, tj. ne postoji mera distance, svi algoritmi koriste jednostavne obilaske grafova kao što su pretraga po dubini (DFS, *depth-first search*) i pretraga po širini (BFS, *breadth-first search*).

Pribavljanje tela metoda koje se pozivaju iz bloka

Da bi kontrola toka mogla da se prati neometano kroz pozive metoda u okviru klase, neophodno je poznavati koje sve funkcije i metode mogu da se pozove iz jednog bloka koda. Blok koda najčešće predstavlja metodu ili funkciju, mada to može biti i blok kao što je jedna od grana u uslovnom grananju korišćenjem izraza *if*.

U bloku za koji se pribavljaju tela vrši se iteracija nad svim potomcima sintaksne vrste *CallExpression* (§3.2.1.3). Među izrazima se traže oni kod koji izraz koji se poziva pripada vrsti *PropertyAccessExpression* (??), i to takav da se na levoj strani nalazi ključna reč *this* (??).

Za izraze koji zadovoljavaju ove uslove beleži se ime pozvane metode. Vodeći se ovim imenom, pronalaze se tela tih metoda. Ona se dodaju u skup rezultata, a od njih proces počinje da se ponavlja rekurzivno.

Pribavljanje imena svojstava kojima se vrednost može promeniti izvršenjem bloka

Kako bi se uvidela veza između modela i šablona, potrebno je razumevanje toga koje metode i kako mogu da utiču na promenu vrednosti svojstva u klasi.

Ovaj algoritam se odvija kroz dve faze. U prvoj fazi se traže imena svojstava kojima se vrednost može promeniti *direktnim* izvršenjem bloka. Drugim rečima, ne prolazi se kroz pozive metoda. Nakon što se prva daza okonča, postupak se rekurzivno ponavlja za sve metode koje se mogu pozvati iz bloka, korišćenjem algoritma opisanog u §5.1.2.1.

Potruga za svojstvima obavlja se prolaskom kroz potomke sintaksne vrste *ExpressionStatement* (??). Od interesa su binarni, prefiksni unarni i postfiksni unarni izrazi.

Za binarne izraze od interesa su samo oni kod kojih je operator jedan od operatora dodele (§3.2.1.4). Ovime se odstranjuju binarni izrazi – na primer, *a < b*. Zatim se odstranjuju izrazi kod kojih leva strana binarnog izraza ne predstavlja

pristup svojstvu (kao `a.b`), a potom se odstranjuju i oni kod kojih leva strana pristupa nije `ThisKeyword`. Preostali pristupi u sebi sadrže ime svojstva kojem se menja vrednost (npr. `this.foo`).

Za prefiksne i postfiksne unarne izraze, pretraga je jednostavnija. Uzimaju se samo oni izrazi kod kojih važi da je operand pristup svojstvu, i to takvih da je prvi pristup ključna reč `this`. Kao i u slučaju binarnih izraza, iz njih se dobija ime svojstva kojem se menja vrednost.

Analizatori komponenti

Osnovne informacije o komponenti, nezavisno od njene konkretne upotrebe u stablu komponenti aplikacije, pribavljaju se iz analizatora komponente. U procesu izgradnje stabla analizatora fabrika, nailaskom na komponentu se najpre kreira njen analizator, pa se onda on koristi za kreiranje analizatora *fabrike* komponente.

Ime komponente

Ime komponente ne igra značajnu ulogu u tokom razvića aplikacije jer se nigde ne koristi. Umesto imena komponente koriste se simboličko ime kojim je komponenta *registrovana*, a ne *definisana*. Ipak, *neko* ime se mora dodeliti komponenti kada se ona umetne u DOM tokom izvršenja.

Mada ime komponente može biti samo proizvoljan jedinstven string, radi preglednijeg rezultujućeg DOM-a sa boljom semantikom, Vejn koristi informacije kojima raspolaže da komponenti dodeli ime.

Ako je definisano ime klase, ono se koristi kao ime komponente. Inače se radi o podrazumevano izvezenom simbolu iz modula, pa se koristi ime modula – osim u slučaju kada je ime modula `index`: tada se koristi ime direktorijma u kome se nalazi modul. Ovako dobijeno ime komponente se za potrebe imena proizvoljnog HTML taga pretvara u *param-case* oblik.

Međutim, HTML standard nalaže da imena proizvoljno definisanih tagova ne samo da moraju da budu mala slova, već da *moraju* sadržati bar jednu crticu kako bi se obezbedila kompatibilnost sa HTML elementima koji će u standard biti dodati u budućnosti (jer se za njih zna da nikad nemaju nijednu crticu). Ukoliko se ime komponente sastoji od dve reči, ovo ne predstavlja problem, ali imena od jedne reči neće moći da se koriste kao ime taga.

Zato se, radi konzistentnosti, uvek dodaje prefiks `w-` ispred imena HTML taga. Na primer, komponenta `Item` dobija tag `w-item`, dok `ListOfItems` dobija tag `w-list-of-items`, iako bi ime sadržalo crtice i bez prefiksa.

Konstante

Obilaskom koda se za svako svojstvo klase može odrediti da li se ono može smatrati svojstvom iz koga se podaci samo čitaju, a nikada ne menjaju. Za ovakvo svojstvo klase se kaže da je **konstanta**.

Preduslov da se svojstvo prolagsi za konstantu jeste da nije ulaz u komponentu – ovo se jednostavno određuje ispitivanjem modifikatora kojim je obeleženo svojstvo (bez modifikatora – implicitno `public`, eksplicitno `public`, `protected`, `private`). Zatim se prolazi kroz sve blokove koji predstavljaju tela metoda klase i pribavljaju se imena svojstava koja se mogu modifikovati izvršenjem tog bloka, na osnovu algoritma opisanog u §5.1.2.2. Ukoliko se ime svojstva za koje se ispituje status konstante ne nađe u tako dobijenom skupu, radi se o konstanti.

Ovo se koristi za optimizaciju opisanu u TODO.

Analizatori fabrika

Analizator fabrike služi da generatoru koda pruži neophodne informacije da bi mogao da obavi svoj zadatak. Nalaze se u čvorovima stabla analizatora fabrika.

Pribavljanje putanje do druge fabrike

Između svake dve fabrike postoji najkraća putanja. Pošto se radi o stablu, najkraća putanja je jedina putanja u kojoj se putanja ne „vraća unazad”, pa je dovoljno koristiti običan DFS ili BFS algoritam za pronalazak ove putanje.

Putanja je predstavljena nizom susednih čvorova u stablu, onim redom kojim ih treba obići da bi se iz jedne fabrike „stiglo” u drugu.

Ova funkcionalnost je važna za generisanje koda koji se tiče prenosa podataka iz jednog čvora u drugi (što se događa prilikom vezivanja vrednosti za šablon).

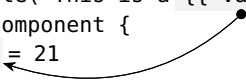
Pribavljanje stvarnog imena vezane vrednosti

Autoru je u šablonima prilično intuitivno na koji opseg se odnose imena vezana za šablon, ali proces obračunavanja *stvarne* vrednosti (ili putanje do vrednosti) je drugačiji, pogotovo kada se uzme u obzir činjenica da se na nekim mestima kod optimizuje (v. §5.1.3.2).

U slučaju **komponenti**, pristupni lanac se najpre razbija na delove. Među svojstvima, geterima i metodama se traži onaj sa istim imenom kao prvi deo lanca. Ukoliko se on ne pronađe, znači da se u njoj ne nalazi tražena referenca. Ovo je znak generatoru, koji poziva ovu metodu, da nastavi traženje dalje uz lanac roditelja duž grane u stablu analizatora, sve dok ne nađe na prvi analizator koji se ne nalazi između definicione fabrike i odgovorne fabrike za čvor u kome

je definisan povež (više u TODO). Ukoliko se u deklaraciji klase pronađe traženi simbol, onda se na osnovu §5.1.3.2 određuje da li se radi o konstanti ili ne.

```
@Template('This is a {{ value }}.')
class Component {
  value = 21
}
```



Kod **uslovnih direktiva**, proces je znatno jednostavniji jer je *nemoguće* da vrednost bude vezana za nju. Ova direktiva ne nudi nikakve simbole koje bi šablon mogao da iskoristi za povezivanje.

Direktive za ponavljanje, s druge strane, pružaju jedan ili dva simbola koji se mogu iskoristiti. To su *iterativna konstanta* (uvek) i *iterativni indeks* (ukoliko se navede). Kao i u slučaju komponenti, lanac se razbija na delove, ali se poredi samo sa jednom ili dve gorenavedene vrednosti. Nijedan ni drugi simbol ne mogu biti konstante. U slučaju pogotka, generatoru se vraća informacija o tome koji iterativni simbol je u pitanju. Ova vrednost se zatim koristi da se pristupi *pravom* imenu (umesto simboličkom koje developer zadaje radi razumevanja koda).

Povezi u pogledu i njihovo mapiranje

Postoji i inverzni način ispitivanja koji daje sličnu informaciju. Umesto da se za konkretan lanac pristupa ispituje da li pripada određenom analizatoru, mogu se dobiti sve instance poveza za koje bi pribavljanje stvarnog imena dalo pogodak.

U ovo spadaju i povezi definisni nad „pravim” HTML elementima i nad komponentama i direktivama koji postoje samo prividno tokom pisanja Vejn aplikacije. Mogu se dobiti i particije ovog skupa po ovoj osobini.

```
@Register(B)
@Template('The value is {{ val1 }} and <B [b]="val2"/>.')
class A { private val1!: any; private val2!: any }
```

U gornjem primeru, za komponentu A, to su

- povež definisan inteprolacijom `{{ val1 }}` (interpolacija se obavlja nad Text čvorom koji je deo HTML-a) i
- ulaz `ba]="val2"` (primenjen nad komponentom B što je Vejn komponenta).

Mape razlika

Od važnosti nije samo *koji* povezi postoje, već i na kojim se elementima nalaze i koja je fabrika odgovorna za njihovo ažuriranje. U ovu svrhu definišu se mape

razlika. Mape razlika sadrže preslikavanje elementa sistema koji se ažurira u niz poveza koje na njima treba ažurirati. Zovu se „mape razlika” jer pokazuju kako treba *mapirati* dobijene *razlike* prilikom pokretanja procesa ažuriranja unutar fabrike. Postoje **mape razlika u DOM-u** i **mape razlika u fabrikama**. Razlikuju se u tome o kom je elementu sistema reč – o postojećim HTML elementima ili o elementima specifičnim za Vejn (tj. o komponentama i direktivama).

```
@Template('
  <div [id]="a" [className]="b">{{ a }}</div>
  <w:for foo of foos>
    <span [className]="a">{{ foo }}</span>
  </w:if>
')
class Component {
  a!: any
  b!: any
  foos!: any[]
}
```

Na primer, u TODO isečku koda definisana je komponenta čija bi mapa razlika u DOM-u definisala sledeće preslikavanje.

- Za čvor `<div>`: `[id]="a"` i `[className]="b"`.
- Za tekstualni čvor u `div-u`: `{{ a }}`.
- Za čvor `` unutar uslovne direktive: `[className]="a"`.

Za razliku od `[className]="a"`, tekstualni čvor `{{ b }}` unutar `span-a` nije deo preslikavanja – simbol `foo` nije definisan u komponenti `Component`, već u direktivi `w: for` kao alijas za trenutni element niza `foos` prilikom iteracije.

S druge strane, mapa razlika u fabrikama bi sadržala samo jedno preslikavanje.

- Za direktivu `w: if`: svojstvo `a` vezano kao uslovna promenljiva.

Indeksi čvorova u šablonima i pogledima

Mape indeksa bile bi beskorisne za generisanje koda ukoliko ne bi bilo poznato kako doći do odgovarajućih instanci tokom izvršenja programa. Pošto postoji jasno definisano uređenje čvorova, svakom je dodeljen indeks u skladu sa BFS obilaskom. Za čvor u šablonu ili pogledu se može pribaviti odgovarajući indeks u slučaju komponente ili odgovarajući indksi u slučaju direktive.

Direktive imaju dva indeksa jer su u realnom DOM-u predstavljene sa dva HTML čvora – tačnije, dva komentara. Više u TODO.

Povezi definisani nad sidrom

Sidro (*anchor*) je čvor pogleda roditeljskom čvoru iz stabla analizatora zbog koje se analizator pojavio u stablu. Drugim rečima, sidro je inverzno mapiranje čvora pogleda na dete-analizator. Sidro se koristi da bi se pristupilo povezima definisanim nad komponentom ili direktivom. U sledećem primeru, sidro za komponentu B je čvor `` definisan u komponenti A.

```
@Template('{{ b }}')
class B { public b!: any }

@register(A)
@Template('<B [b]="val"/>')
class A { private val!: any }
```

Analizator sa sidra može dovući listu svih poveza koji tu deklarirani. U prethodnom primeru, to je ulaz `[b]="val"`.

Jedna od primena ovoga jeste pronalazak toga koji izlazi se koriste nad komponentom i koju metodu poziva u komponenti u kojoj je sidro definisano (v. TODO).

Određivanje fabrika na koje se može uticati izvršenjem bloka

Osim određivanja metoda koje se mogu pozvati unutar jedne komponente, važno je odrediti i koje metode se mogu pozvati *izvan* nje. Jedini način da metoda komponente pozove metodu druge komponente jeste ukoliko su komponente povezane izlazom.

Nakon što se na osnovu algoritma §5.1.2.1 pronađu metode koje mogu biti pozvane određenim blokom, traže se one metode koje su obeležene kao javne, jer to znači da se mogu iskoristiti kao izlazi. Zatim se pomoću §5.1.4.6 filtriraju metode koje zaista jesu iskorišćene kao izlaz, i pamte se imena metoda koja se pozivaju u komponenti-pretku.

Za ovako dobijene metode traži se koja se unija svojstava koji se mogu promeniti u toj komponenti (na osnovu §5.1.2.2). Za istu komponentu se traže i sva svojstva i/ili geteri definisani u klasi koja je upisuje (na osnovu §5.1.4.3). Ukoliko postoji presek između ova dva skupa, znači da se pozivom metode iz originalne komponente može javiti promena na pogledu roditeljske komponente. Ovo generator koda koristi kako bi se odredilo za koje fabrike treba pozvati metode ažuriranja (v. §5.3.1.8).

Postupak se zatim rekurzivno nastavlja, jer se iz komponente-pretka mogu dalje pozivati metode koje predstavljaju izlaze, koje su u nekom daljem pretku vezane za neku drugu metodu, itd. Po završetku obilaska, dobija se skup svih fabrika na čije poglede jedna metoda može uticati.

Proces generisanja koda

U fazi generisanja koda izdvaja se pet glavnih celina:

- priprema,
- preprocesiranje komponenti,
- generisanje fabrika,
- postprocesiranje komponenti i
- bandlovanje.

Priprema

U fazi pripreme generišu se neki generički fajlovi sa pomoćnim funkcijama koje se mogu koristiti tokom ostalih faza generisanja koda.

Preprocesiranje komponenti

U fazi preprocesiranja komponenti menja se kôd klase napisan od strane autora. Pored vršenja transformacija koda, informacije o promenama dostavljaju se analizatoru kako bi dopunile dostupne informacije.

Obavljene transformacije tiču se **asinhronih** delova koda. Naime, najčešće se promene u modelu dešavaju kao posledica pokretanja neke akcije korisnika tokom njegovog interagovanja sa korisničkim interfejsom. Međutim, postoje još dva inicijatora: tajmeri i mreža.

U Javaskriptu se tajmeri postavljaju korišćenjem globalno dostupnih funkcija `setTimeout` i `setInterval`. Mrežni zahtevi se mogu obaviti pomoću `XMLHttpRequest`-a ili `fetch`-a.

Ubrizgavanje fabrike

Da bi se omogućilo da se pogled ažurira nakon izvršenja asinhronog dela koda, potrebno je da se pozove metoda koja vrši njegovo osvežavanje. Međutim, metoda za osvežavanje nalazi se u fabrici komponente, a klasa nema pristup njoj. Zato je prvo neophodno da se obezbedi da referenca postoji.

Referenca se prosleđuje kroz konstruktor. Ukoliko on postoji, dodaje se novi argument; inače se kreira konstruktor sa praznim telom i argumentom kojim se čuva referenca na fabriku.

```
// pre
class Foo { }

// posle
class Foo {
  constructor (private factory)
}
```

Normalizacija i umetanje koda

Obilaskom AST-a klase kojom je deklarirana komponenta traže se instance poziva funkcije `setTimeout` i funkcije `setInterval`, kao i pojave poziva metoda `.then` i `.catch` (nad instancama klase `Promise`). Sve ove funkcije i metode kao prvi argument primaju funkciju koju treba izvršiti kada se dogodi odgovarajući asinhroni događaj.

Postoji nekoliko načina na koji se može proslediti ova funkcija. To može biti referenca na funkciju, a funkcija se može i direktno deklarirati; pri tome za drugi slučaj postoje dva načina: korišćenjem ključne reči `function` ili kao „streličastu” funkciju, korišćenjem tokena `=>`. Streličasta funkcija takođe ima više varijanti – zagrade oko argumenata su opcione ako postoji samo jedan argument (a nije mu specificiran Tajpskript tip), a zagrade za telo funkcije i ključna reč `return` su takođe opcione u nekim slučajevima.

Zbog velikog broja načina na koji se funkcija može definisati, pre nego što se u kod umetne poziv funkcije za ažuriranje pogleda, vrši se **normalizacija funkcije**. Funkcija se transformiše tako da se radi o streličastoj funkciji (ovime se izbegavaju greške ukoliko autorova funkcija koristi `this` koje treba da pokazuje na klasu, a ne na funkciju kreiranu ovom transformacijom), a autorov kod se pretvara u strukturu nalik na IIFE¹. Ovime je dobijen ekvivalentan kod, ali je napisan u obliku u kome je lako dodati sporedne efekte koji se neće mešati sa namerom koju ima autor funkcije.

```
(...args: any[]) => {
  const result = (/* autorov kod */)(...args)
  /* umetnut kod */
  return result
}
```

Svaki asinhroni deo koda se obeležava nenegativnim celim brojem koji predstavlja indeks elementa u nizu. Umetnut kod poziva metodu iz fabrike koja se bavi ažuriranjem pogleda, pri čemu se pomenuti broj koristi za pristup potrebnoj metodi.

```
/* umetnut kod */
this.factory.updateAsync[0]()
```

¹*Immediately Invoked Function Expression*, često korišćeni obrazac u Javaskriptu gde se funkcija koja se definiše odmah i izvršava; na primer, `((() => 21)())`.

Generisanje fabrika

Fabrike se generišu u posebnim datotekama, obilaskom stabla analizatora fabrika. U ovoj fazi čvorovi više nisu međusobno zavisni, pa redosled obilaska nije od važnosti. Za svaki tip analizatora fabrika postoji odgovarajući generator koda koji se koristi.

O samoj strukturi fabrika govori se u §5.3, ??, §5.4 i §5.5.

Postprocesiranje komponenti

U fazi postprocesiranja, autorov kod se dodatno obrađuje kako bi se pripremio za krajnji produkt.

Uklanjanje dekoratora

Kada autor koristi dekoratore u kodu, oni služe za kompilatoru dostave neke specifične informacije kako bi se u fazi generisanja fabrika generisao odgovarajući kod. Sami dekoratori nemaju nikakav uticaj na izvršenje koda.

Zapravo, kada autor uveze simbole kao `Template` iz modula `wane`, uvoze se samo Tajpskript deklaracije. Kod dekoratora koji bi se izvršavao kada se aplikacija pokrene uopšte **ne postoji**. Ako bi se kod pokrenuo kao takav, došlo bi do greške `ReferenceError` jer se simboli kao `Template` ne bi pronašli. Zato se svi dekoratori u potpunosti brišu.

Izvoz klasa

Svaka klasa definisana od strane autora prebacuje se u posebnu datoteku koja se uvozi u odgovarajuće fabrike. Da bi se obezbedilo da je uvoz moguć, osim pukog kopiranja teksta koji čini deklaraciju klase, mora se dodati i ključna reč `export`, ukoliko ona ne postoji.

Uklanjanje konstanti

Na osnovu rezultata algoritma opisanog u §5.1.3.2, poznato je koja se svojstva klase mogu ukloniti iz deklaracije klase i prebaciti u datoteku u kojoj se nalaze sve konstante.

Pored pukog uklanjanja svojstava sa klase, važno je obraditi i delove koda gde se konstanta koristi i uvesti modul sa konstantama kako bi se njoj moglo pristupiti.

```
// pre
class Component {
  private constant = 21
```



```

    private variable = 42
    private get sum () {
        return this.constant + this.variable
    }
    private mutate (val) {
        this.variable = val
    }
}

// posle
Component$constant = 21
class Component {
    private variable = 42
    private get sum () {
        return Component$constant + this.variable
    }
    private mutate (val) {
        this.variable = val
    }
}

```

Ukoliko se ispostavi da se konstanta koristi samo jednom (ili dovoljno malo puta), minifikator može dodatno skratiti ovaj kod tako što će svako pojavljivanje pristupa konstanti zamenisi samom konstantom (v. §5.2.5).

Bandlovanje

Nakon što se sve datoteke generišu, sledi vaza bandlovanja. Kao i ceo proces generisanja koda, i ovo je implementacioni detalj i ne treba da se tiče autora koda. Trenutno se koristi Rolap (*Rollup*) za sklapanje svih datoteka u jednu, ali se ovaj izbor može promeniti u bilo kom trenutku, i kôd autorā neće morati da pretrpi nikakve promene, čak ni u konfiguracionim podešavanjima.

U toku bandlovanja vrši se i minifikacija. Za minifikaciju koristi se Terzer (*terser*)² u vidu plagina za Rolap.

Jedna od najznačajnijih faza u procesu minifikacije pored uklanjanja znakova beline jeste menglovanje (*mangling*). Mada se imena u velikom broju slučajeva mogu bez problema promeniti, to nije slučaj sa imenima svojstva. Na primer, u sledećem isečku koda³ se imena `longVariableName` i `sum` skraćuju na `n i e`, kao i simbolička imena argumenata `first` i `second` (ponovo na `n i e`).

```

// ulaz
const longVariableName = {
    oneNumber: 21,

```

²Ranije Aglifaj-E-Es (*UglifyES*), najpopularniji alat za minifikaciju Javaskript modula.

³Novi redovi u izlazu su dodati radi preglednost. Izlaz Terzera nema nove redove jer bez potrebe doprinose veličini koda.

```

    anotherOne: 3,
  }
  function sum (first, second) {
    return first + second
  }
  console.log(sum(
    longVariableName.oneNumber,
    longVariableName.anotherOne))

// izlaz
const n={oneNumber:21,anotherOne:3};
function e(n,e){return n+e}
console.log(e(n.oneNumber,n.anotherOne));

```

Međutim, imena `oneNumber` i `anotherOne` nisu skraćena. Terzer ne može da pretpostavi da je ova operacija sigurna jer se određeni deo koda može oslanjati na ovo ime – to može biti deo javnog API-ja ili se možda takav format očekuje u JSON obliku sa servera.

Da bi se ova imena skratila, potrebno je eksplicitno uključiti ovo podešavanje. Čak i kada se ono uključi, „poznata” imena kao `createElement` neće biti promenjena. Ako bi se ona promenila, pristup bilo kakvom javnom API-ju u okruženju (npr. `window` u brauzerima) ne bi bio moguć; na primer, `document.createElement('p')` bi postalo `document.e('p')` što nije moguće izvršiti. Međutim, kako Terzer ne zna za tipove podataka, *svako* korišćenje imena `createElement` za pristup svojstvu će ostati nepromenjen, čak iako se ne radi o toj metodi definisanoj nad `document`; na primer, `myVar.createElement()` bi postalo `e.createElement()` jer Terzer ne vrši proveru tipa (i nema načina za to) pa ne zna da li je `e` tipa `Document`.

Kako bi se obezbedilo bolje menglovanje imena svojstava u kodu (što nije uvek sigurna operacija prilikom minifikacije), sva interna imena koja Vejn generiše dobijaju prefiks `__wane__`. Sva imena koja počinju tako, iako su ime svojstva, dobijaju kraće ime u procesu minifikacije.

Anatomija fabrika komponenti

Fabrika komponente je funkcija koja vraća objekat. Nad objektom su definisane neke metode koje se pozivaju tokom životnog ciklusa komponente, kao i neka svojstva u koja se smešta stanje komponente koje se koristi prilikom izvršenja metoda.

Svaka fabrika ima najviše četiri glavne metode:

- `init`⁴ – poziva se jednom, tokom inicijalizacije komponente,

⁴Metoda `init` se u kodu generiše kao metoda `__wane__init`, kao i sve ostale metode i svojstva

- `diff` – poziva se posle inicijalizacije i svaki put u okviru metode za ažuriranje pogleda,
- `update` – poziva se svaki put kada se primeti da je došlo do potencijalne promene u modelu, i
- `destroy` – poziva se jednom, kada komponentu treba uništiti i ukloniti iz DOM-a.

U zavisnosti od rezultata analize, neke metode se mogu izbaciti i time se smanjuje kod aplikacije.

Metoda `init`

Metoda `init` služi da kreira neophodne DOM čvorove i inicijalizuje instancu klase deklarisanu od strane autora, kao i daprvim pozivom metode za traženje razlike `diff` omogućiti ažuriranje komponente.

Koren komponente

Koren komponente je čvor u DOM-u za koji se komponenta vezuje – koreni čvorovi pogleda vezuju se za koren kao njegova deca. Smešta se u promenljivu

U slučaju da se radi o pristupnoj komponenti (§4.2.2), za koren se koristi `body element`.

```
this.root = document.body
```

U svim ostalim slučajevima, koren se uzima iz svojstva `domNodes` roditeljske fabrike (v. §5.3.1.6), navođenjem odgovarajućeg indeksa. Indeks se dobija pribavljanjem sidra (v. §5.1.4.6) i određivanjem njegovog indeksa (v. §5.1.4.5).

```
this.root = this.factoryParent.domNodes[0]
```

Instanca klase

U instanci klase nalaze se metode koje vrše manipulaciju nad stanjem ili ispaljuju događaje na koje precizno komponente u stablu mogu da se pretplate definisanjem poveza za njene izlaze. Fabrici je potrebna ova instanca kako bi bilo moguće odrediti razliku u stanju tokom procesa ažuriranja u metodu `diff` (v. §5.3.2), kako bi se mogle čitati vrednosti, i kako bi mogle da se pozivaju metode definisane od strane autora.

U najjednostavnijem slučaju se klasa jednostavno instancira korišćenjem operatora `new`.

koja se pominju u ovom i drugim odeljcima. Prefiks `__wane__` koristi se da bi se omogućila minifikacija (v. §5.2.5). Ovde se metode navode bez prefiksa radi lakšeg čitanja i pisanja.

```
this.data = new Component()
```

Međutim, u slučaju da je potrebno ubrizgati fabriku u komponentu radi asinhronih ažuriranja (v. §5.2.2.1), konstruktor se može pozvati i sa argumentom `this`.

```
this.data = new Component(this)
```

Prihvatanje podataka kroz ulaze i izlaze

U komponentu se mogu proslediti podaci pomoću ulaza i mogu se osluškivati događaji pomoću izlaza.

Kada se kreira instanca klase koja predstavlja komponentu, pribavljaju se ulazni podaci od fabrika-predaka. Pribavljanje se obavlja jednostavnim operatorom dodele.

```
// deo šablona  
<Component [a]="b"/>
```

```
// generisan kod  
this.data.a = this.factoryParent.b
```

Mada sintaksno izlazi izgledaju drugačije od ulaza, i mada se nameće drugačija semantika, izlazi se interno tretiraju vrlo slično ulazima. Zapravo, jedini razlog zbog koga se ne tretiraju apsolutno identično u slučaju da nema argumenata jeste ponašanje ključne reči `this` u Javaskriptu. Zato se umesto puke dodele uvek pravi streličasta funkcija.

```
// deo šablona  
<Component (a)="b()"/>
```

```
// generisan kod  
this.data.a = () => {  
  this.factoryParent.b()  
}
```

U slučaju da su pozivu funkcije u okviru poveza prosleđeni argumenti, i oni se obračunavaju kao i sva ostala svojstva i metode.

```
// deo šablona  
<Component (a)="b(c)"/>
```

```
// generisan kod  
this.data.a = () => {  
  this.factoryParent.b(this.factoryParent.c)  
}
```

Ukoliko se kao argument nađe `#`, on se prosleđuje kao argument funkcija koja se dodeljuje izlazu.

```
// deo šablona
<Component (a)="b(#)" />

// generisan kod
this.data.a = (placeholder) => {
  this.factoryParent.b(placeholder)
}
```

Kreiranje DOM čvorova

Svi DOM čvorovi koji su deklarirani šablonskom sintaksom u sklopu komponente instantiraju se i smeštaju u niz kako bi im se kasnije moglo pristupiti.

Za njihovo instanciranje koriste se pomoćne metode koje samo delegiraju zadatak HTML metodama `createElement`, `createTextNode` i `createComment` definisanim nad tipom `Document` umesto direktnog pristupa. Ovime je omogućeno da se za njihova imena veže prefiks `__wane__` i time obavi minifikacija ovih imena (v. §5.2.5).

```
this.domNodes = [
  /*0*/ createTextNode('No entered data.'),
  /*1*/ createElement('button'),
  /*2*/ createTextNode('Add some'),
];
```

Sastavljanje DOM čvorova

Sada kada elementi postoje, treba ih sastaviti tako da oponašaju autorovu definiciju kroz šablone. Za ovo se takođe jednostavne pomoćne funkcije. Stablo pogleda se obilazi rekursivno.

```
appendChild(this.root, [
  this.domNodes[0],
  appendChildren(this.domNodes[1], [
    this.domNodes[2],
  ])
]);
```

Registrowanje dece

Registrowanje dece se obavlja pomoćnom funkcijom `createFactoryChildren`. Osim što se za tekuću fabriku vezuju prosledjena deca-fabrike, vrši se i povezivanje u obrnutom smeru – deci se kao roditelj dodeljuje tekuća fabrika.

```
createFactoryChildren(this, [
  ChildComponentFactory(),
])
```

Ukoliko fabrika nema decu i ukoliko fabrika nema `destroy` metodu (v. §5.3.4), ovaj deo koda se ne generiše. Ukoliko metoda `destroy` postoji, neophodno je da postoji prazan niz `this.factoryChildren = []` kako bi rekurzivni obilazak prilikom uništavanja radio.

Inicijalizacija svojstava i atributa u DOM-u

Iako su HTML čvorovi kreirani, u opštem slučaju i dalje u potpunosti ne odgovaraju onome što je autor definisao jer nisu dodeljene vrednosti atributima i svojstvima u skladu sa deklaracijom šablona.

Koristeći indekse za pristup DOM čvorovima (v. §5.1.4.5), štampaju se odgovarajuće dodele (za svojstva) i pozivi metoda (za atribute). Ovo uključuje i obične deklaracije (`id="a"`) i poveze (`[id]="a"`).

```
this.domNodes[3].type = 'text'
this.domNodes[3].className = 'field'
this.domNodes[3].setAttribute('aria-label', 'Name')
```

Pretplata na događaje iz DOM-a

Za pretplatu na DOM događaje koristi se pomoćna metoda koja delegira poziv standardnoj metodi `addEventListener`. Osim poziva funkcije koju je autor definisao kroz šablon, dodaje se i niz poziva metodā `update` nad svim fabrikama nad kojim poziv autorove metode može uticati u skladu sa algoritmom definisanim u §5.1.4.7.

```
addEventListener(this.domNodes[1], 'submit', (placeholder) => {
  this.data.onSubmit(placeholder)
  this.factoryParent.factoryParent.factoryParent.update()
  this.factoryParent.update()
});
```

Deklaracija metoda za asinhrono ažuriranje

Slično kao kod pretplate, generišu se pozivi metodā `update`, s tim što se u ovom slučaju smeštaju u niz kako bi se određena metoda pozvala iz samog koda klase (v. §5.2.2.1).

```
this.updateAsync = [
  /* 0 */ () => { this.update() },
  /* 1 */ () => { this.factoryParent.update() },
];
```

Ako asinhronog koda u komponenti nema, ovaj deo koda se u potpunosti izostavlja.

Prva razlika

Kako bi se pokrenuo proces računanja razlika između „prethodnog” i trenutnog stanja, mora se populisati `prevData` odgovarajućim vrednostima. Najbrži način da se ovo uradi je prost poziv metode `diff` (v. §5.3.2).

U specijalnim slučajevima kada komponenta nema stanje (ili su svi njegovi elementi proglašeni kao konstantni, v. §5.1.3.2), ovaj deo koda se ne generiše jer tada metoda `diff` ne postoji.

Primer

Primer čitave metode `init` dat je u nastavku.

```
init() {
  // §5.3.1.1: Koren komponente
  this.root = this.factoryParent.domNodes[0]

  // §5.3.1.2: Instanca klase
  this.data = new EmptyState()

  // §5.3.1.3: Prikupljanje podataka kroz ulaze i izlaze
  this.data.createNew = () => {
    this.factoryParent.factoryParent.factoryParent
      .data.onCreateNewClick()
  };

  // §5.3.1.4: Kreiranje DOM čvorova
  this.domNodes = [
    /*0*/ createTextNode('\n No entered data. '),
    /*1*/ createElement('button'),
    /*2*/ createTextNode('Add some!'),
  ]

  // §5.3.1.5: Sastavljanje DOM čvorova
  appendChildren(this.root, [
    this.domNodes[0],
    appendChildren(this.domNodes[1], [
      this.domNodes[2],
    ]),
  ])

  // §5.3.1.6: Registrovanje dece
  this.factoryChildren = [];
```

```

// §5.3.1.7: Inicijalizacija svojstava i atributa u DOM-u
// Nema

// §5.3.1.8: Pretplata na događaje iz DOM-a
addEventListener(this.domNodes[1], 'click', () => {
  this.data.createNew();
  this.factoryParent.factoryParent.factoryParent.update();
})

// §5.3.1.9: Deklaracija metoda za asinhrono ažuriranje
// Nema

// §5.3.1.10: Prva razlika
this.diff();
}

```

Metoda diff

Metoda za traženje razlika jednostavno vraća objekat sa razlikama. U pitanju je objekat tipa `Record<keyof C, boolean>`, gde je `C` klasa pridružena komponenti. Drugim rečima, u pitanju je objekat kome su ključevi imena svojstava deklariranih u klasi (odnosno neki njihov podskup), a vrednosti su `true` ili `false`.

Računanje razlike oslanja se na prethodnu vrednost stanja sačuvanu u `prevData` i poredi vrednosti sa trenutnom vrednošću sačuvanu u `data`. Zahvaljujući jednom pokretanju funkcije `diff` „u prazno” (rezultat se odbacuje), ovaj pristup funkcioniše i prilikom prvom pokretanju ove metode.

Računanje razlike i ažuriranje prethodne vrednosti obavlja se istovremeno.

```
value: this.prevData.value !== (this.prevData.value = this.data.value),
```

Najpre se izvršava leva strana poređenja (`this.prevData.value`), i ta vrednost se poredi sa rezultatom izvršenja operatora `=` s desne strane (`this.prevData.value = this.data.value`). Taj rezultat je vrednost `this.data.value`, pa se poređenje vrši između prethodne i trenutne vrednosti. Sporedni efekat primena operatora `=` je to što je sada u `this.prevData.value` upisana trenutna vrednost. Kako se tokom jednom poziva funkcije `diff` ista vrednost neće koristiti dvaputa, ovaj pristup funkcioniše.

Metoda se koristi u okviru metode `update` (v. §5.3.3). Ne generiše se ukoliko nema stanja za koje ima potrebe računati razliku.

Metoda update

Pre nego što se započne sa bilo kakvim ažuriranjem, preračunava se razlika trenutnog stanja u odnosu na prethodno. Ovo se obavlja pozivom metode `diff` (v. §5.3.2). Rezultat se čuva u lokalnoj konstanti `diff`.

Za generisanje koja koji vrši ažuriranje koriste se mape razlika (v. §5.1.4.4).

Ažuriranje DOM elemenata

Iz mape razlika čitaju se indeksi DOM čvorova za koje su vezani povezi. Zatim se vrši uslovno ažuriranje na osnovu imena svojstva iz klase i rezultata traženje razlika.

```
if (diff.title) {
  this.domNodes[2].data = this.data.title;
}
if (diff.value) {
  this.domNodes[3].value = this.data.value;
}
```

Ažuriranje potomaka

Slično kao kod DOM elemenata, za fabrike se takođe dobavlja indeks i skup poveza. Nakon obavljenog ažuriranja, poziva se metoda `update` nad fabrikom čije je stanje ažurirano.

Da bi se obezbedilo da se ažuriranje deteta obavlja samo jedno, vrši se grupisanje uslova operatorom `||`.

```
if (diff.visibleItems || diff.items) {
  this.factoryChildren[0].data.items = this.data.visibleItems;
  this.factoryChildren[0].data.totalItemsCount = this.data.items.length;
  this.factoryChildren[0].update();
}
```

Metoda destroy

`destroy` je jednostavna metoda koja najpre poziva ovu metodu za decu (kako bi se proces uništenja dogodio rekurzivno, počev od fabrika koje su listovi), a zatim prolazi kroz sve DOM elemente koji su direktni potomci korena i uklanja ih.

```
destroy() {
  this.factoryChildren.forEach(factoryChild => {
    factoryChild.destroy()
  });
}
```

```

while (this.root.firstChild !== null) {
  this.root.removeChild(this.root.firstChild)
}
}

```

Fabrike uslovne direktive (**w:if**)

Fabrike uslovne direktive sastoje se iz dva dela. Jedan deo se stara za to da se sadržaj unutar direktive stvori i uništi u trenutku kada se primeni vrednost vezana za **w:if**, a drugi deo se stara da sadržaj koji se stvori bude onaj sadržaj koji je autor definisao.

Zajednički deo se naziva **uslovni pogled** (*conditional view*), a deo specifičan za svaku instancu direktive naziva se **parcijalni pogled** (*partil view*).

Uslovni pogled

Komponente imaju koreni čvor unutar koga se kreira njen sadržaj. Ovaj čvor slući da nedvosmisleno definiše početak i kraj komponente. Kada komponentu treba uništiti, jasno je određeno koji čvorovi treba da budu oklonjeni iz DOM-a.

U slučaju direktiva, proizvoljan skup DOM čvorova može biti obuhvaćen direktivom. Ne samo da se može javiti više HTML elemenata, već neki čvorovi uopšte ne moraju da budu elementi – mogu biti tekstualni čvorovi.

Kako bi se znalo odakle počinje i gde se završava sadržaj direktive, u DOM se ubacuju dva **oluka** (*outlet*). Oluk služi da usmeri delove koda na to gde se tačno određeni deo pogleda (parcijalni pogled) nalazi. Prvi oluk (po redosledu dokumenta) naziva se **početni** a drugi **krajnji**.

Četiri metode se definišu za uslovne poglede: metoda za inicijalizaciju, metoda za inicijalizaciju parcijalnog pogleda, metoda za ažuriranje i metoda za uništenje.

Metoda **init**

Prilikom inicijalizacije uslovnog pogleda, slično komponentama, najpre se dobavlja koren – samo je u ovom slučaju to oluk koga čine dva komentara. Umesto dobavljanja jednog DOM čvora iz roditelja, dobavljaju se dva – jedan je početni komentar, a drugi krajnji.

```

this.openingCommentOutlet = this.factoryParent.domNodes[4]
this.closingCommentOutlet = this.factoryParent.domNodes[5]

```

Za razliku od komponenti, gde se data fabrike određuje na osnovu autorovog koda (v. §5.3.1.2), kod direktiva se unapred zna kog je oblika stanje. U slučaju

w:if, to je jedna logička vrednost. Ona se čuva u promenljivu data. Prethodno stanje čuva se u promenljivoj prevData, isto kao i kod fabrika komponenti. U metodi init se ova dva svojstva inicijalizuju na vrednost koja im je prosleđena.

```
this.data = this.prevData = !this.factoryParent.data.isEmpty
```

Zatim se, u slučaju da je vrednost this.data istinita, odmah poziva metoda za inicijalizaciju parcijalnog pogleda. Inače se proces inicijalizacije okončava. Sadržaj uslovne direktive pojaviće se kada se this.data promeni u istinitu vrednost pokretanjem metode update.

```
if (this.data) {  
    this.initPartialView()  
}
```

Metoda initPartialView

Za inicijalizaciju samog pogleda unutar direktive, tj. parcijalnog pogleda koji je vezan za direktivu, koristi se metoda initPartialView. Kao prvo i jedino dete tekuće fabrike inicijalizuje se parcijalni pogled i prosleđuju mu se oluci. Zatim se inicijalizuje sâm parcijalni pogled pozivom njegove metode init.

```
this.factoryChildren[0] = PartialView_ConditionalView_isEmpty_5_4()  
this.factoryChildren[0].factoryParent = this  
this.factoryChildren[0].openingCommentOutlet = this.openingCommentOutlet  
this.factoryChildren[0].closingCommentOutlet = this.closingCommentOutlet  
this.factoryChildren[0].init()
```

Metoda update

Ažuriranje uslovnog pogleda svodi se na ispitivanje koja od četiri moguće kombinacije vrednosti svojstava prevData i data je u pitanju.

Kada su obe vrednosti false, znači da se direktiva nalazila u „isključenom” stanju i da ostaje u njemu. U ovoj situaciji ne treba uraditi ništa.

Prelazak iz false u true znači da je direktiva bila „isključena” i da se ovim ažuriranjem ona „uključuje”. Ovaj slučaj se obrađuje tako što se pokrene metoda initPartialView kojom se sadržaj direktive kreira i dodaje u DOM.

Prelazak iz true u false znači da je direktiva bila „uključena” i da se ovim ažuriranjem ona „isključuje”. Tada se parcijalni pogled treba uništiti, pa se poziva ova metoda.

Obe vrednosti true znači da samo treba ažurirati parcijalni pogled.

```
update(diff) {  
    const prev = this.prevData;
```

```

    this.prevData = this.data;
    if (!prev && !this.data)
        return;
    if (!prev && this.data)
        return this.initPartialView();
    if (prev && !this.data)
        return this.factoryChildren[0].destroy();
    this.factoryChildren[0].update(diff);
}

```

Metoda destroy

Metoda za uništavanje delegira zadatak parcijalnom pogledu, ukoliko on postoji u trenutku kada se ova metoda pozove.

```

destroy() {
    if (this.factoryChildren[0] != null) {
        this.factoryChildren[0].destroy();
    }
}

```

Parcijalni pogled

Parcijalni pogled po mnogo čemu podseća na komponentu. Može se reći i da je komponenta je specijalni slučaj parcijalnog pogleda. Glavna razlika je u tome što ne postoji data svojstvo. Svi podaci koji se mogu iskoristiti u parcijalnom pogledu dolaze iz komponenti i direktiva (i to ne `w:if`) koje se nalaze iznad tekućeg čvora u stablu fabrika.

Pored toga, razlikuje se način vezivanja kreiranih DOM čvorova za postojeći DOM, ali samo u prvom nivou. Pošto koren nije jedan element, umesto `appendChild` koristi se `insertBefore` nad krajnjim olukom. U skladu sa ovime razlikuje se i `destroy` metoda.

Primer celog parcijalnog pogleda dat je u nastavku.

```

var PartialView_ConditionalView_isEmpty_5_4 = () => ({
    prevData: {},
    init() {
        this.domNodes = [
            /*0*/ createElement('button'),
            /*1*/ createTextNode('Save'),
            // ...
        ]
        insertBefore(this.closingCommentOutlet, [
            appendChildren(this.domNodes[0], [
                this.domNodes[1],

```

```

        // ...
    })
  })
  this.factoryChildren = []
  this.domNodes[0].type = 'submit'
},
diff() {
  return {
    // ...
  }
},
update(diff) {
  Object.assign(diff, this.diff())
  // ...
},
destroy() {
  this.factoryChildren.forEach(child => child.destroy())
  let node = this.openingCommentOutlet.nextSibling
  while (node !== this.closingCommentOutlet) {
    const nextNode = node.nextSibling
    node.remove()
    node = nextNode
  }
},
})
})

```

Fabrike direktive za ponavljanje (w: for)

Konceptualno, direktiva za ponavljanje funkcioniše isto kao i uslovna direktiva – sastoji se iz dva dela, pri čemu se unutrašnji deo takođe zove **parcijalni pogled** i ima identičnu strukturu kao i parcijalni pogled vezan za uslovnu direktivu, a spoljni deo **ponavljajući pogled** (*repeating view*).

Struktura ponavljajućeg pogleda je, međutim, znatno drugačija od strukture uslovnog pogleda. Sadrži najdinamičniji deo koda iz celog frejmworka; zbog prirode direktive nemoguće mnogo stvari odrediti statički jer je nemoguće predvideti kako će se dodavati elementi u niz, brisati iz njega, i kako će menjati mesta. Ovo je verovatno najneoptimalniji deo generisanog koda, pa se u poređenju sa ostatakom generisanog koda može učiniti kao previše glomazan, ali se zapravo bazira na konceptima traženje razlike između dva niza i sličan je algoritmima za *dom diff* koji koriste mnoge biblioteke. Konkretno je inspirisan funkcijom za traženje i „krpljenje” razlika iz jedne od ranijih verzija mikro-frejmvorka *Hyperapp* [25].

TODO: objasni algoritam, trebaće neke jednostavne slike... nabrzaka: ideja je da se prolazi kroz oba niza paralelno u nadi da će da se pronađu odgovarajući

elementi (to znači da im je isti ključ) u starom nizu kada se nađe na novi niz. na elementu u novom nizu se stoji sve dok se ne nađe na isti element u starom nizu, pri čemu se u hešu (tj javaskript objektu što nije baš $O(1)$ ali je dovoljno blizu) čuvaju svi običeni elementi da bi lako moglo da im se pristupi kasnije (da ne moramo da prolazimo kroz niz dvaput). kada prođemo kroz ceo stari niz, ako nema tog elementa kog tražimo, kreiramo ga. sada barem imamo sve elemente u hešu i lako je da im pristupimo pa samo cepamo kroz novi niz i ili radimo move elementa kad je pogodak ili ga stvorimo kad je pronašaj. sve vreme pratimo gde se trenutno nalazimo u DOM-u (ne samo gde smo u novom i starom nizu) da bismo znali da insertBeforeujemo na mesto gde treba. poslednji element ne ide insertBefore nad nekim elementom nego ide na closing outlet celog w:fora. malo je nezgodno da se celo objasni onako striktno ali uz jednostavne primere koji ilustruju par ključnih slučajeva, neki edge case i jedan-dva kompletna primera koji ilustruju ceo obilazak i rezonovanje, i naravno pre svega toga idejno rešenje bi trebao da bude dovoljno jasno

Stilovi

Kao što je već pomenuto u §4.4, stilovi se enkapsuliraju, tj. vezani su samo za komponentu kojoj su pridruženi. Ovo je implementirano tako što se svim HTML čvorovima dodeljuje isti atribut, jedinstven za svaku komponentu, a CSS se transformiše tako da se uz selektore doda taj atribut.

```
// autorov kod
@Template('
  <span className="one">
    Some <strong>text</strong> here
  </span>
  <span className="two">
    And <strong>more</strong>
  </span>
')
@Style('
  span { color: red }
  .one { text-style: italic }
  span.two > strong { text-decoration: underline }
')
class Example { }

// DOM tokom izvršenja
<w-example>
  <span className="one" data-w-1>
    Some <strong data-w-1>text</strong> here
  </span>
  <span className="two" data-w-1>
    And <strong data-w-1>more</strong>
```

```

    </span>
</w-example>

```

```

// Stilovi tokom izvršenja
span[data-w-1] { color: red }
.one[data-w-1] { text-style: italic }
span.two[data-w-1] > strong[data-w-1] {
    text-decoration: underline
}

```

Osim ovoga, transformiše se i ime komponente.

```

// autorov kod
@register(Child)
@Template('
    <span>Text</span>
    <Child/>
')
@Style('
    span { background-color: red }
    Child { background-color: blue }
')
class Example { }

// DOM tokom izvršenja
<w-example>
    <span data-w-1>Text</span>
    <w-child data-w-1><!-- ... --></w-child>
</w-example>

// Stilovi tokom izvršenja
span[data-w-1] { background-color: red }
w-child[data-w-1] { background-color: blue }

```

Da bi se ovo obezbedilo, tokom inicijalizacije svojstava i atributa u DOM-u (v. §5.3.1.7) se dodaju i atributi oblika `data-w-N`, gde je `N` jedinstven broj za svaku komponentu.

```

this.domNodes[0].setAttribute('data-w-1', '')
this.domNodes[2].setAttribute('data-w-1', '')
this.domNodes[3].setAttribute('data-w-1', '')

```


Literatura

- [1] Addy et al. *Yet Another Framework Syndrome (YAFS)*. Feb. 3, 2014. URL: <https://medium.com/tastejs-blog/yet-another-framework-syndrome-yafs-cf5f694ee070> (visited on 08/31/2018).
- [2] Jose Aguinaga. *How it feels to learn JavaScript in 2016*. Oct. 3, 2018. URL: <https://hackernoon.com/how-it-feels-to-learn-javascript-in-2016-d3a717dd577f> (visited on 08/31/2018).
- [3] Ben Aston. *A brief history of JavaScript*. Apr. 1, 2015. URL: <https://medium.com/@benastontweet/lesson-1a-the-history-of-javascript-8c1ce3bffb17> (visited on 08/31/2018).
- [4] Mihai Bazon. *UglifyJS – JavaScript parser, compressor, minifier written in JS*. URL: <http://lisperator.net/uglifyjs/> (visited on 08/31/2018).
- [5] Tim Berners-Lee. *Answers for Young People*. URL: <https://www.w3.org/People/Berners-Lee/Kids.html#invent> (visited on 08/30/2018).
- [6] Tim Berners-Lee. *HTML tags*. Nov. 3, 1992. URL: <https://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html> (visited on 08/30/2018).
- [7] Tim Berners-Lee. *The WorldWideWeb browser*. URL: <https://www.w3.org/People/Berners-Lee/WorldWideWeb.html> (visited on 08/31/2018).
- [8] Matt Brophy. *A Look Inside Vue's Change Detection*. Nov. 15, 2017. URL: <https://brophy.org/post/a-look-inside-vues-change-detection> (visited on 08/31/2018).
- [9] Christopher Buecheler. *A brief, incomplete*. Sept. 11, 2017. URL: <https://closebrace.com/articles/2017-09-11/a-brief-incomplete-history-of-javascript> (visited on 08/31/2018).
- [10] Giamir Buoncristiani. *What is React Fiber?* Apr. 23, 2017. URL: <https://giamir.com/what-is-react-fiber> (visited on 08/31/2018).
- [11] *CoffeeScript*. URL: <https://coffeescript.org/> (visited on 08/31/2018).
- [12] CoffeeScript contributors. *What is the future of coffeescript?* URL: <https://github.com/jashkenas/coffeescript/issues/4288> (visited on 08/31/2018).
- [13] Dart contributors. *Dart's Type System*. URL: <https://www.dartlang.org/guides/language/sound-dart> (visited on 08/31/2018).

- [14] MDN contributors. *Control flow*. Mozilla. URL: https://developer.mozilla.org/en-US/docs/Glossary/Control_flow (visited on 08/24/2018).
- [15] MDN contributors. *CSS preprocessor*. Mozilla. URL: https://developer.mozilla.org/en-US/docs/Glossary/CSS_preprocessor (visited on 08/25/2018).
- [16] MDN contributors. *Getting started with the Web*. Mozilla. URL: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web (visited on 08/31/2018).
- [17] MDN contributors. *String.prototype.includes()*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/includes (visited on 08/31/2018).
- [18] Rollup contributors. *Rollup: Guide*. URL: <https://rollupjs.org/guide/en> (visited on 08/31/2018).
- [19] TypeScript contributors. *TypeScript*. URL: <https://github.com/Microsoft/TypeScript>.
- [20] TypeScript contributors. *Using the Compiler API*. TypeScript. URL: <https://github.com/Microsoft/TypeScript/wiki/Using-the-Compiler-API> (visited on 08/30/2018).
- [21] Wikipedia contributors. *Abstract syntax tree* — Wikipedia, The Free Encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=849296366 (visited on 08/30/2018).
- [22] Douglas Crockford and Tom Weißbeckel. *Douglas Crockford: „Java was a colossal failure; JavaScript is succeeding because it works”*. Dec. 24, 2012. URL: <https://jaxenter.com/douglas-crockford-java-was-a-colossal-failure-javascript-is-succeeding-because-it-works-105395.html> (visited on 08/31/2018).
- [23] Brendan Eich. Komentar na sajtu *Hacker News*. Mar. 25, 2015. URL: <https://news.ycombinator.com/item?id=9266517> (visited on 08/31/2018).
- [24] Ian Elliot. *Death of Flash and Java Applets*. July 14, 2015. URL: <https://www.i-programmer.info/news/86-browsers/8783-death-of-flash-and-java.html> (visited on 08/31/2018).
- [25] *Funkcija patch u mikro-frejmorku Hyperapp*. URL: <https://github.com/lazarljubenovic/hyperapp/blob/3f7bc8029ae1f080a9c4c916668a3c816735ef2b/src/app.js#L171>.
- [26] Marcio Galli, Roger Soares, and Ian Oeschger. *Inner-browsing extending the browser navigation paradigm*. May 16, 2003. URL: https://developer.mozilla.org/en-US/docs/Archive/Inner-browsing_extending_the_browser_navigation_paradigm (visited on 08/31/2018).
- [27] Jesse James Garret. *Ajax: A New Approach to Web Applications*. Feb. 18, 2005. URL: <http://adaptivepath.org/ideas/ajax-new-approach-web-applications/> (visited on 08/31/2018).
- [28] Naomi Hamilton. *The A-Z of Programming Languages: JavaScript*. July 31, 2008. URL: <https://www.computerworld.com.au/article/255293/a-z-programming-languages-javascript/> (visited on 08/31/2018).

- [29] Marijn Haverbeke. *Eloquent JavaScript, 3rd edition*. 2018. URL: <http://eloquentjavascript.net/index.html> (visited on 08/31/2018).
- [30] Miško Hevery. *<angular/>: A Radically Different Way of Building AJAX Apps*. July 29, 2010. URL: <http://misko.hevery.com/2010/07/29/a-radically-different-way-of-building-ajax-apps/> (visited on 08/31/2018).
- [31] Miško Hevery. *Hello World, <angular/> is here*. Nov. 28, 2009. URL: <http://misko.hevery.com/2009/09/28/hello-world-angular-is-here/> (visited on 08/31/2018).
- [32] Hayato Ito. *Shadow DOM*. Mar. 1, 2018. URL: <https://www.w3.org/TR/shadow-dom/> (visited on 08/25/2018).
- [33] *jQuery Usage Statistics*. Aug. 31, 2018. URL: <https://trends.builtwith.com/javascript/jquery> (visited on 08/31/2018).
- [34] Tab Atkins Jr. and Erika Etemad. *CSS Scoping Module Level 1*. Apr. 2014. URL: <https://www.w3.org/TR/shadow-dom> (visited on 08/25/2018).
- [35] Byung-Keun Kim. *Internationalizing the Internet. The Co-evolution of Influence and Technology*. Edward Elgar Publishing, 2005. ISBN: 9781843764977.
- [36] Marcel Laverdet. *XHP: A New Way to Write PHP*. Feb. 10, 2010. URL: <https://www.facebook.com/notes/facebook-engineering/xhp-a-new-way-to-write-php/294003943919/> (visited on 08/31/2018).
- [37] Niels Leenheer. *make your pages load faster by combining and compressing javascript and css files*. Dec. 18, 2006. URL: <http://rakaz.nl/2006/12/make-your-pages-load-faster-by-combining-and-compressing-javascript-and-css-files.html> (visited on 08/31/2018).
- [38] Kim Maida. *How to Manage JavaScript Fatigue*. Mar. 22, 2017. URL: <https://auth0.com/blog/how-to-manage-javascript-fatigue/> (visited on 08/31/2018).
- [39] Addy Osmani. *Front-end Choice Paralysis*. Jan. 19, 2014. URL: <https://the-pastry-box-project.net/addy-osmani/2014-January-19> (visited on 08/31/2018).
- [40] Tero Parviainen. *Change And Its Detection In JavaScript Frameworks*. Mar. 2, 2015. URL: <http://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html> (visited on 08/31/2018).
- [41] Tero Parviainen. *Overcoming JavaScript Framework Fatigue*. July 15, 2015. URL: <https://teropa.info/blog/2015/07/15/overcoming-javascript-framework-fatigue.html> (visited on 08/31/2018).
- [42] Miłosz Piechocki. *Change detection in Angular versus AngularJS*. Nov. 8, 2017. URL: <https://codewithstyle.info/change-detection-angular-versus-angularjs/> (visited on 08/31/2018).
- [43] Pascal Precht. *Angular Change Detection Explained*. Dec. 18, 2016. URL: <https://blog.thoughtram.io/angular/2016/02/22/angular-2-change-detection-explained.html> (visited on 08/31/2018).
- [44] Tom Preston-Werner. *TOML. Tom's Obvious, Minimal Language*. URL: <https://github.com/toml-lang/toml> (visited on 08/25/2018).

- [45] Dr. Axel Rauschmayer. *Exploring ES2016 and ES2017*. URL: <http://exploringjs.com/es2016-es2017.html> (visited on 08/31/2018).
- [46] Dr. Axel Rauschmayer. *Exploring ES2018 and ES2019*. URL: <http://exploringjs.com/es2018-es2019/index.html> (visited on 08/31/2018).
- [47] Dr. Axel Rauschmayer. *Exploring ES6*. URL: <http://exploringjs.com/es6.html> (visited on 08/31/2018).
- [48] Dr. Axel Rauschmayer. <http://2ality.com/2015/11/tc39-process.html>. URL: <http://2ality.com/2015/11/tc39-process.html> (visited on 09/11/2018).
- [49] Dr. Axel Rauschmayer. *JavaScript for impatient programmers*. URL: <http://exploringjs.com/impatient-js/index.html> (visited on 08/31/2018).
- [50] Dr. Axel Rauschmayer. *Speaking JavaScript*. O'Reilly. URL: <http://speakingjs.com/> (visited on 08/31/2018).
- [51] Sacha, Raphaël, and Michael. *The State of JavaScript (2017). Survey Results*. URL: <https://stateofjs.com/2017/> (visited on 08/25/2018).
- [52] Victor Savkin. *Change Detection in Angular*. July 13, 2016. URL: <https://vsavkin.com/change-detection-in-angular-2-4f216b855d4c> (visited on 08/31/2018).
- [53] stunix. *The story of www.slashdotslash.com*. Apr. 2012. URL: <http://stunix.outanet.com/?p=slash> (visited on 08/31/2018).
- [54] Aaron Swartz. *A Brief History of Ajax*. Dec. 22, 2005. URL: <http://www.aaronsw.com/weblog/ajaxhistory> (visited on 08/31/2018).
- [55] Basarat Ali Syed and contributors. *TypeScript Deep Dive*. URL: <https://legacy.gitbook.com/book/basarat/typescript> (visited on 08/30/2018).
- [56] Vincent Tunru. *A short history of Javascript Frameworks: a comparison of JQuery, AngularJS and React*. June 28, 2016. URL: <https://vincenttunru.com/A-short-history-of-Javascript-frameworks-a-comparison-of-JQuery-AngularJS-and-React/> (visited on 08/31/2018).
- [57] Angular University. *Angular Change Detection – How Does It Really Work?* July 18, 2018. URL: <https://blog.angular-university.io/how-does-angular-2-change-detection-really-work/> (visited on 08/31/2018).