UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Aleksandar Milosavljević

RAZVOJ OKRUŽENJA "PURE" ZA RAZVOJ VEB INTERFEJSA

master rad

Beograd, 2021.

Mentor:

dr Saša Malkov, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Aleksandar Kartelj, docent
Univerzitet u Beogradu, Matematički fakultet

dr Ivan Čukić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mojim roditeljima.

Naslov master rada: Razvoj okruženja "Pure" za razvoj veb interfejsa

Rezime: U okviru ovog rada predstavljen je razvoj okruženja *Pure* - modernog okruženja za razvoj veb aplikacija baziranog na šablonu *Flux*. U narednim poglavljima nalaze se obrazloženja za odabir konkretnih alata i biblioteka tokom implementacije, primeri korišćenja okruženja za pisanje korisničkih aplikacija kao i objašenje implementacionih detalja najvažnijih elemenata okruženja.

Ključne reči: veb, internet, interfejs, html, css, JavaScript, TypeScript, razvojno okruženje, React, Angular, Redux, Flux, MVC

Sadržaj

Glava 1

Uvod

Iako današnji mini računari, koje gotovo svi nosimo u džepu, imaju više procesorske snage nego računar koji je korišćen tokom Appolo misije [12], danas te lične računare ne koristimo za teška izračunavanja i komplikovane procedure. Koristimo ih najčešće kao jednostavne terminale kako bismo pristupili internet resursima, to jest, podacima koji se nalaze na velikim centralizovanim računarskim sistemima. Veb interfejs je tako postao glavni način naše interakcije sa računarom i internetom.

Posledica toga je značajna promena fokusa u svetu razvoja aplikativnog softvera. Ukoliko pogledamo rezultat upitnika kompanije *„Stack Overflow"* o najkorišćenijim programerskim alatima, videćemo da prva dva mesta na listi zauzimaju upravo veb tehnologije (Na prvom mestu je JavaScript sa 69.7%, dok je na drugom HTML/CSS sa 62.4%) [26].

Kako potreba za novim softverom daleko prevazilazi mogućnost softverskih inženjera da taj softver isporuče, biblioteke i okruženja koja pomažu pri razvoju softvera postali su veoma važan deo skupa inženjerskih alata. U domenu razvoja veb aplikacija, na tržištu se izdvajaju tri razvojna okruženja. Ova tri razvojna okruženja najviše se međusobno razlikuju po pristupu u promeni stanja aplikacije i razumevanju toga šta je centralni deo dizajna (u funkcionalnom smislu) jedne veb aplikacije.

Kada posmatramo ova popularna razvojna okruženja, uočavamo dva bitno različita pristupa u arhitekturi:

1. Šablon *Model-Pogled-Upravljač* u kom aplikacija predstavlja skup povezanih komponenti organizovanih u hijerarhisku strukturu, kojima je pridruženo stanje i ponašanje.

2. Šablon *Flux* u kom se aplikacija posmatra kao mašina stanja čija je vizuelna reprezentacija samo rezultat trenutnog stanja.

Prvi pristup je prisutan u okruženju *Angular*, drugi je dominantan u biblioteci *React*, dok okruženje *Vue.js* zastupa dizajn koji predstavlja kompromis između ova dva pristupa.

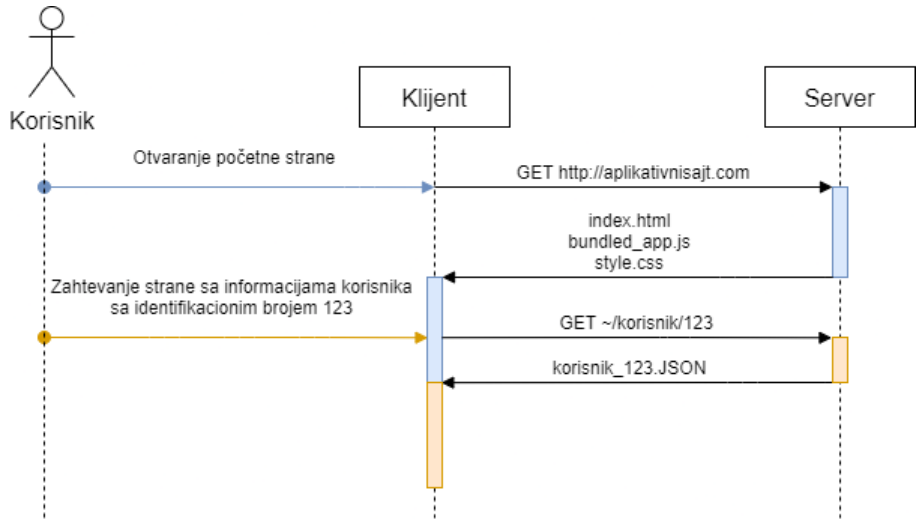
1.1 Organizacija rada

Kroz ovaj rad ispratićemo razvoj jednog modernog razvojnog okruženja za izradu veb aplikacija. U poglavlju 2 opisaćemo arhitekturu SPA, kao najčešći arhitekturalni šablon korišćen pri izradi veb aplikacija. Zatim ćemo se u poglavlju 3 osvrnuti na najčešće arhitekturalne šablone klijentskih aplikacija kroz analizu problema najpopularnijih okruženja i objasniti motivaciju za pisanje novog razvojnog okruženja. U poglavlju 4 definisaćemo formalne zahteve koje bi novo okruženje trebalo da ispuni. Nakon toga, u poglavlju 5, upoznaćemo se sa novorazvijenim okruženjem za razvoj SPA aplikacija iz perspektive korisnika (inženjera). U poglavlju 6 ćemo se upoznati sa implementacionim detaljima samog orkuženja. U poglavlju 7 daćemo objašenje na koji način su ispunjeni zahtevi postavljeni u poglavlju 4 i analizirati novorazvijeno rešenje iz perspektive budućih proširenja i unapređenja.

Glava 2

Arhitektura SPA

Poboljšanjem infrastrukture Interneta i napredovanjem tehnologije i mogućnosti ličnih računara i mobilnih uređaja, prezentaciona logika korisničkih aplikacija se postepeno selila sa centralizovanih serverskih računara na klijentske uređaje. Arhitektura SPA (eng. *Single Page Application*) je dovela ovaj princip do krajnjih granica prebacivanjem celokupne logike ponašanja aplikacije na klijentsku stranu [7]. U ovoj arhitekturi klijent pri prvom zahtevu ka serveru preuzima celokupni (klijentski) aplikativni kod i obrađuje ga u pregledaču, dok svi naredni pozivi služe prostoj razmeni podataka. Ovo je značajno uticalo na samu metodologiju razvoja veb aplikacija.



Slika 2.1: Dijagram sekvence SPA arhitekture.

Ovaj pristup omogućio je razdvajanje poslova timova zaduženih za serversku logiku i timova zaduženih za prezentacionu logiku i vizuelni dizajn, a time omogućio

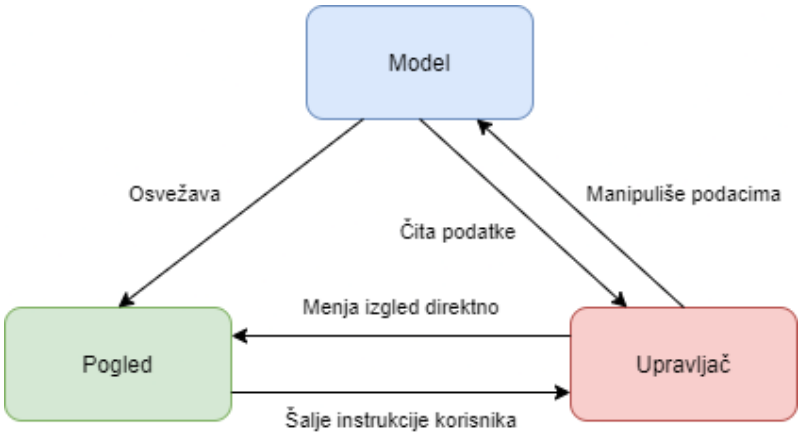
i veći stepen njihove nezavisnosti. Jedina tačka spajanja ova dva tima postao je dizajn interfejsa između klijentske i serverske aplikacije. Rešenje ovog problema inženjeri su našli u arhitekturalnom šablonu REST (eng. *Representational State Transfer*) [5].

Arhitekturalni šablon REST standardizovao je način organizacije podataka i način pristupanju podacima kroz API (eng. *Application Programmable Interface*). Dizajniran je oko podataka (resursa), koji su organizovani po hijerarhijskom modelu strukture fajl-sistema.

Odvajanjem razvoja klijentskih i serverskih aplikacija, pisanje složenih korisničkih aplikacije postalo je značajno lakše. Povećavanjem složenosti razvoja ovih aplikacija, razvila se potreba za razvojnim šablonima i razvojnim okruženjima koja omogućavaju programerima da lakše organizuju svoj kod i da lakše njime upravljaju. Prvi šabloni koji su se koristili preuzeti su iz arhitekture višestraničnih aplikacija. Primer jednog takvog šablona je *Model-Pogled-Upravljač* (eng. *Model-View-Controller*) [1].

2.1 Šablon Model-Pogled-Upravljač

Šablon *Model-Pogled-Upravljač* nastao je pre pojave arhitekture SPA. Koristio se za velike sajtove i aplikacije, ali na takav način da su sve komponente pisane kao serverski kod. Ideja šablona *Model-Pogled-Upravljač* je da omogući razdvajanje odgovornosti između logike podataka, aplikativne logike i vizuelne reprezentacije programa.

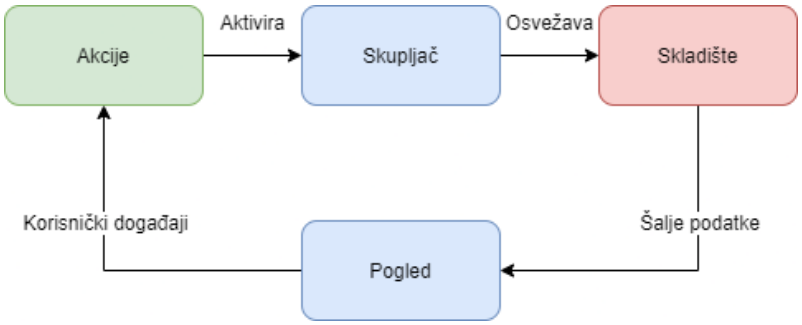


Slika 2.2: Arhitektura šablona Model-Pogled-Upravljač

Ovaj šablon ima sličnu primenu i u SPA aplikacijama, sa tom razlikom što se sve odvija na klijentu. Umesto apstrakcije nad bazom podataka, *model* predstavlja apstrakciju nad podacima učitanim u aplikaciju. *Upravljač* (ili kontroler) je zadužen za aplikativnu logiku i za prihvatanje instrukcija poslatih sa nivoa *pogleda*. Sloj *pogleda* služi za opis vizuelne reprezentacije aplikacije i ne sadrži nikakvu aplikativnu logiku. Najpoznatiji predstavnik okruženja koja koriste ovaj šablon je *Angular* [9].

2.2 Šablon Flux

Šablon *Flux* je nastao u kompaniji *Facebook*, iz potrebe inženjera da otklone probleme sa deljenjem stanja između različitih komponenti i čestim nadmetanjem za resurse (eng. *race condition*) [19].



Slika 2.3: Arhitektura Šablona Flux

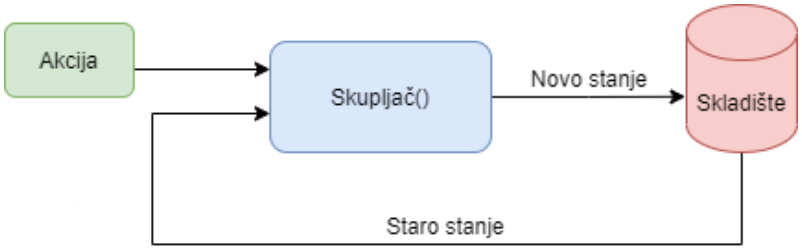
Osnovna ideja ovog šablona je da podaci teku u jednom smeru, tako da se uvek može jednostavno utvrditi redosled akcija i na taj način sprečiti nadmetanje za resurse između različitih komponenti. Način na koji šablon *Flux* rešava problem sa nadmetanjem za resurse sastoji od činjenice da se *JavaScript* kod izvršava uvek u jednoj procesorskoj niti i modelu izmene stanja u kojem su sve operacije koje menjaju stanje atomične[1].

Pogled, kao i kod šablona *Model-Pogled-Upravljač*, služi za opis vizualne reprezentacije programa. *Pogled* utiče na tok podataka tako što pri korisničkom događaju odabire *akciju* koju će da emituje ka *skupljaču* (eng. *reducer*).

Skupljač je čista funkcija [21], koja kao argumente prihvata akciju i trenutno stanje skladišta i vraća novo stanje skladišta. Povratna vrednost skupljača se potom upisuje u skladište umesto starog stanja, tako da ne postoji slučaj deljenja

[1]Atomično - Nedeljiva celina. Atomična operacija se izvršava po principu „sve ili ništa".

iste reference objekta između dve komponenente. Sloj pogleda jedne komponente može samo da pročita stanje skladišta, ali ne i da ga menja neposredno, već na njega može da utiče isključivo okidanjem neke akcije. Nakon obrade akcije, skladište okruženja sadrži novi objekat i tako zna da mora da osveži iscrtavanje DOM-a.



Slika 2.4: Skupljač

Važno je naglasiti da šablon *Flux* za razliku od šablona *Model-Pogled-Upravljač* ne insistira na podeli slojeva po fajl strukturi na takav način da dva sloja ne mogu biti deo istog fajla. Razdvajanje odgovornosti se ovde posmatra u domenu toka podataka i toga koji sloj može upravljati i pristupati podacima u *kom trenutku*. Biblioteka *React* nastala je u kompaniji *Facebook* kao predstavnik ovog šablona [8].

Glava 3

Motivacija

Iako su moderna razvojna okruženja poput okruženja *Angular* i biblioteke *React* omogućila programerima da napišu veoma kompleksne korisničke aplikacije na mnogo lakši način nego što je to ranije bio slučaj, to ne znači da ova okruženja nemaju svoje mane i specifične zamke. Ovde ćemo pomenuti neke od njih.

3.1 Problemi sa okruženjem Angular

Okruženje *Angular* popularno je među programerima koji razvijaju softver za korporativne korisnike. Razvojni timovi koji rade na razvoju ovakvog softvera često broje preko 20 inženjera i testera. Šablon *Model-Pogled-Upravljač* koji prestavlja osnovni arhitekturalni dizajn okruženja Angular, značajno olakšava rad velikom broju ljudi na istom programskom kodu.

3.1.1 Upravljanje stanjem

Problem sa okruženjem *Angular* nastaje kada među različitim komponentama moramo da delimo stanje. Nadmetanje za resursima vremenom počinje da se javlja sve češće kao problem, a „pametna krpljenja" počinju da zauzimaju sve veći deo programskog koda.

Problem sa deljenjem stanja, pored problema sa nadmetanjem za resurse, je i taj što aplikacija postaje previše kompleksna za razumevanje čak i za iskusnije inženjere i problemi u kodu se vremenom sve bolje sakrivaju, čekajući na krajnje korisnike.

3.1.2 Logika u pogledu

Pogledajmo jedan primer koda sa sajta `https://angular.io/`:

```
<div *ngFor="let item of items">{{item.name}}</div>
```

Primer koda 3.1: Primer koda u *Angular*-u

Kod prikazan u primeru 3.1 predstavlja instrukciju za izlistavanje stavki (`item`) iz niza `items`. Drugim rečima, ovo nije HTML[1]-element, već instrukcija napisana na specijalnom jeziku okruženja *Angular*, namenjenom za opisivanje pogleda, koja će u vreme izvršavanja biti zamenjena pravim HTML-elementom. Ovaj HTML-element će dinamički promeniti svoj sadržaj svaki put kada se sadržaj niza `items` promeni.

Ovo je jednostavan primer, ali nije teško zamisliti situaciju u kojoj programska logika postaje previše kompleksna da bi se o njoj rezonovalo u slučaju potrage za greškom. Da stvar bude gora, ovu logiku nije moguće (jednostavno) debagovati[2] modernim alatima ugrađenim u pregledače, jer logika ispisana u ovoj instukciji prolazi kroz više faza prevođenja i uglavnom je na kraju izvršava optimizovani i minifikovani kod samog okruženja, koji je veoma nečitljiv.

3.2 Problemi sa bibliotekom React

Okruženje *React* razvijeno je od strane *Facebook* inženjera sa ciljem da stvore okruženje dizajnirano tako da šablon *Flux* bude lak za implementaciju pri razvoju aplikacija. Ovaj model okruženja omogućio je u velikoj meri izbegavanje problema nadmetanja za resurse, ali su neki drugi problemi ipak ostali deo rešenja.

3.2.1 Razdvajanje odgovornosti

U primerima 3.2 i 3.3 vidimo primer preuzet sa zvanične veb stranice biblioteke *React*. U pitanju su dva različita načina definisanja komponente u okruženju *React*. U primeru 3.2 je prikazan način koji koristi sintaksu *JSX*. *JSX* je skraćenica za *JavaScript-XML*, što može da se uoči i iz prikazanog primera. Ideja je da se komponenta može u potpunosti opisati u jednom fajlu sa ekstenzijom `.jsx` (ili `.tsx` ako koristite *TypeScript* [10]).

[1] Hyper-Text-Markup-Language.
[2] *Debug* - eng. Traženje i uklanjanje grešaka.

Umesto posebnog fajla koji bi koristio HTML-sintaksu (ili sintaksu nalik HTML-sintaksi, kao što je to slučaj u okruženju *Angular*), u biblioteci *React*, struktura komponente može se opisati na istom mestu gde i ponašanje komponente. Ovo često omogućava lakše rezonovanje o povezanosti vizualne strukture komponente i njenog ponašanja i predstavlja pristup razdvajanja odgovornosti na nivou komponenenata, a ne na nivou tehnologije.

Ovaj mehanizam se u u pozadini ostvaruje tako što React transpilira[3] ovaj kod na čist *JavaScript* (Primer 3.3). Korisnik može svoje komponente pisati i na način prikazan u primeru 3.3, ali je primer 3.2 očigledno lakši za čitanje i rezumevanje strukture.

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

Primer koda 3.2: JSX-sintaksa

```
const element = React.
    createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

Primer koda 3.3: HTML-sintaksa

Iako je ovo veoma zgodan mehanizam, on uvodi neke neželjene posledice. Naime, jako je teško utvrditi šta je ispravan, a šta neispravan *JSX* kod i gde su tačno granice gde počinje HTML, a završava se *JavaScript*. To možemo uočiti i na primeru 3.2. Element `<h1>` u ovom primeru ima postavljenu vrednost atributa `className` na nisku `„greeting”`. Ovaj atribut zapravo se pri prevođenju na HTML, tumači kao `class`. Razlog zašto ne koristimo reč `class` u *JSX* kodu je taj što je `class` rezervisana reč u *JavaScript*-u.

Ovo je samo jedan dobro poznat primer koji zbunjuje nove korisnike, ali sličnih, a bolje sakrivenih primera ima dosta. Ovaj sistem opisa vizuelne strukture podleže gotovo istim problemima kao i *Angular*-ov jezik za opis strukture (eng. *Templating language*), jer nad njim ne možemo koristi napredne alate za debagovanje.

[3]Transpilacija - prevođenje sa jednog jezika na drugi, viši programski jezik.

3.2.2 Okruženje ili biblioteka

Iako *React* nije razvojno okruženje u punom smislu te reči, već biblioteka koja se bavi isključivo opisom sloja pogleda, mnogi inženjeri ga koriste u sklopu ekosistema kao deo razvojnog okruženja. Sve ostale funkcionalnosti koje su neophodne za kompletno razvojno okruženje deo su drugih biblioteka (*ReactDOM, Redux, MobX, ...*). Ovo dovodi do velike heterogenosti u pristupu pisanja koda i zbog toga nema jedinstvenog pristupa rešavanju većini problema koji se javljaju pri razvoju aplikacija. Ovo je upravo razlog zbog kojeg je *Angular* u velikim korporacijskim okruženjima i timovima i dalje dominantan izbor.

3.3 Motivacija pri kreiranju novog okruženja

Motivacija pri pravljenju novog okruženja sastoji se iz želje za ispitivanjem alternativnog pristupa pravljenju jednog okruženja. Novo okruženje bi trebalo da počiva na temeljima dobrih aspekata postojećih rešenja, kombinujući ih tamo gde to ima smisla. U delovima dizajna u kojima ni jedno od postojećih orkuženja nema adekvatan odgovor na predstavljene probleme, novo okruženje bi trebalo da ponudi novo i originalno rešenje problema.

Ukratko, novo okruženje treba da bude moderan alat za razvoj klijentskih veb aplikacija koje u svom dizajnu ne sadrži opisane greške iz predstavljenih rešenja.

Glava 4

Koncept

Kako bismo imali jasnu ideju kako bi novo okruženje, koje rešava opisane probleme, trebalo da izgleda, ovde ćemo definisati motivaciju i formalne zahteve koje bi novo okruženje trebalo da ispuni.

4.1 Zahtevi

Definišimo sada formalne zahteve:

1. Kako bi izbegli probleme iz predstavljenih okruženja kao što su *logika u pogledu* (Pododeljak 3.1.2) i *razdvajanje odgovornosti* (Pododeljak 3.2.1) novo okruženje ne sme da uvodi novi, niti da koristi bilo kakav postojeći specijalizovani jezik za opisivanje strukture HTML-a. Dodatni zahtev u pogledu opisa HTML-strukture komponenenata je omogućavanje debagovanja logike i strukture izgleda komponente alatima koji su ugrađeni u moderne pregledače.
2. Kako bi se izbegao problem sa deljenjem stanja (Odeljak 3.1.1), novo okruženje bi trebalo da se oslanja na šablon *Flux* i jednosmeran tok podataka. Stanje mora biti centralizovano, a interfejs za upravljanje stanjem mora biti robustan i univerzalan za sve komponente.
3. Kako bi što manje grešaka došlo do korisnika, potrebno je uočiti što više grešaka koje je moguće uočiti u fazi razvoja kroz podršku striktnih tipova.
4. Distribucija razvojnog okruženja mora biti dostupna kroz javni repozitorijum *npm*-paketa.

5. Okruženje mora biti otvoreno za modifikovanje. Kako u smislu prilagođavanja okruženja konkretnom projektu, tako i u smislu predloga za modifikovanje samog okruženja kroz *zajednicu otvorenog koda* (eng. *Open Source community.*).
6. Komponente okruženja moraju imati jednostavnu i lako čitljivu strukturu.

4.2 Arhitektura i dizajn okruženja

Novo okruženje preuzeće šablon *Flux* od biblioteke *React*, ali smernice za konretnu implementaciju ovog šablona biće preuzete od biblioteke *NgRx* koja je deo *Angular*-ekosistema [25]. Od okruženja *Angular* takođe će biti preuzet pristup razvoju koji podrazumeva pisanje komponenti u jeziku *TypeScript* umesto jezika *JavaScript*.

4.2.1 Zašto ne HTML?

Iako je HTML oproban jezik za opisivanje strukture dokumenta, on ima svoja ograničenja koja nije tako lako prevazići. Programi za obradu teksta pri pisanju HTML fajla uglavnom teško mogu da zaključe kontekst povezivanja pojedinačnih HTML-elemenata sa konkretnim funkcijama i promenljivim napisanim u *JavaScript*-u i zbog toga je podrška alata, a samim tim i iskustvo tokom pisanja HTML koda, značajno slabije nego pri pisanju *JavaScript* ili *TypeScript* koda. Dodatna mana je u tome što je jako teško segmentirati HMTL kod u hijerarhijsku strukturu ili ga podeliti na više fajlova, jer u HTML-u ne postoji opcija za uključivanje podkomponenti. Korišćenjem *TypeScript* funkcija umesto sintakse HTML-a rešavaju se ovi problemi. Alati za upravljanje *TypeScript* kodom značajno su napredniji od alata za upravljanje HTML-om.

4.3 Ime okruženja

Ime okruženja *Pure*[1] inspirisano je funkcionalnom paradigmom i čistim funkcijama [21]. Funkcionalni elementi (koji će biti opisani u ovom poglavlju), predstavljaju uslovno rečeno čiste funkcije, koje prihvataju trenutno stanje kao argument i vraćaju DOM-element kao rezultat. Ovo nam omogućava da implementiramo i

[1] *Pure* - eng. Čisto, čist.

mehanizam memoizacije (eng. *Memoization*) [21], koji je iskorišćen kao mehanizam optimizacije.

Drugi razlog za ime *Pure* dolazi iz činjenice da okruženje koristi isključivo *JavaScript* za opis strukture DOM-drveta i ponošanja, pa ime *Pure* možemo shvatiti i u konteksu *Pure JavaScript* iliti *Čist JavaScript*.

Glava 5

Ilustracija upotrebe okruženja Pure

U ovom poglavlju upoznaćemo se detaljnije sa novim okruženjem i pogledati par jednostavnih primera aplikacija napisanih u okruženju *Pure*. Krenućemo od najjednostavnijeg primera, aplikacije „Zdravo svete" (eng. *„Hello world"*). Potom ćemo pojasniti organizacionu strukturu aplikacije, a na samom kraju poglavlja ćemo se osvrnuti i na nešto kompleksniji primer *Pure*-aplikacije.

5.1 Preduslovi

Kako bismo mogli da pokrenemo *Pure*-aplikaciju neophodno je da radno okruženje *Node.js* bude instalirano na računaru [23]. Preporučena verzija u vreme pisanja ovog rada je `14.17.5`. (LTS[1]). Pored okruženja Node.js, neophodno je da na računaru bude instaliran i *git*, Pored ovih neophodnih alata, preporučuje se i korišćenje modernog programa za obradu teksta sa podrškom za jezik *TypeScript* [10]. Autor preporučuje alat otvorenog koda, razvijen od strane Microsoft inženjera: *Visual Studio Code* [11].

Okruženje *Pure* može se preuzeti preko *npm*-sistema[2] [24]. To možemo uraditi na dva načina:

1. Instaliranjem paketa `pure-framework` u već postojeći *npm*-modul. (Konzolna komanda: 5.1)

[1] *Long Term Support* - eng. Dugoročna tehnička podrška.
[2] *Node Package Manager* - eng. Upravljač Paketa za Node.

2. Pokretanjem skripte za pravljenje projekta `hello-world`, bez prethodnog instaliranja *npm*-paketa. (Konzolna komanda: 5.2)

5.2 Zdravo svete!

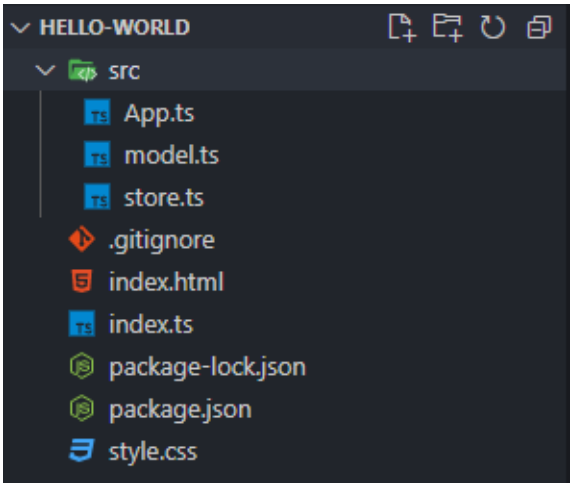
Ukoliko već imamo postojeći *npm*-modul na kojem radimo, možemo u okviru njega instalirati `pure-framework` komandom:

```
$ npm install pure-framework                                    (5.1)
```

Ovde ćemo prikazati jednostavniji pristup u kojem krećemo sa praznim direktorijumom, u kom ćemo pomoću *npx*-skripte napraviti početnu aplikaciju „Zdravo svete" [20]. Potrebno je da se u terminalu pozicioniramo u prazan direktorijum i da odatle pokrenemo komandu:

```
$ npx pure-framework hello-world                                (5.2)
```

Izvršavanje ove komande napraviće novi direktorijum sa nazivom `hello-world` i u njega klonirati repozitorijum[3] sa minimalnom *Pure*-aplikacijom. Ukoliko je skripta izvršena bez grešaka, direktorijum `hello-world` bi trebalo da sadrži fajlove prikazane na slici 5.1.



Slika 5.1: Struktura direktorijuma nakon izvršenja komande 5.2

[3]Repozitorijum: `https://github.com/maleksandar/pure-framework-hello-world-app`.

Kako bismo pokrenuli našu aplikaciju, potrebno je da se pozicioniramo u novonapravljeni direktorijum i instaliramo neophodne *npm*-pakete, izvršavanjem komande:

```
$ cd hello-world && npm install
```
(5.3)

Nakon što se ova komanda uspešno izvrši, možemo da pokrenemo našu aplikaciju komandom:

```
$ npm run start
```
(5.4)

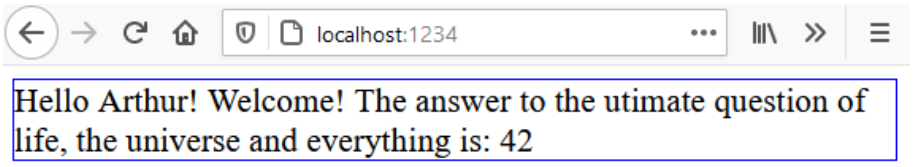
Ukoliko pokretanje ove komande nije izbacilo grešku, konzola bi trebalo da nas obavesti da je server za razvoj pokrenut i da možemo da mu pristupimo na adresi `http://localhost:1234` (Konzolni odgovor 5.1)

```
> pure-framework-hello-world-app@1.0.0 start
> parcel index.html

Server running at http://localhost:1234
Built in 27ms
```

Konzolni odgovor 5.1: Konzolna poruka nakon pokretanja aplikacije

Ukoliko u pregledaču otvorimo stranicu `http://localhost:1234`, trebalo bi da vidimo stranicu koja izgleda kao na slici 5.2



Slika 5.2: Aplikacija „Zdravo svete”, pokrenuta na lokalnom serveru.

5.2.1 Struktura aplikacije

Fajl koji sadrži centralnu logiku naše aplikacije je `src/App.ts` (Primer 5.2).

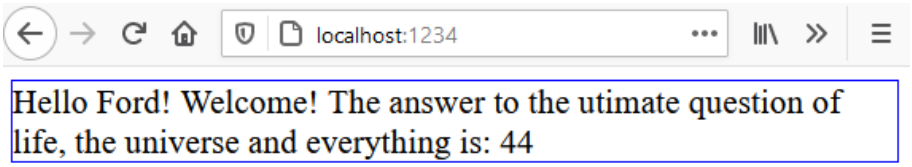
```
1  import { Component, componentFactory } from "pure-framework/core";
2  import { div, span } from "pure-framework/html";
3  import { AppModel } from "./model";
4
5  class AppComponent extends Component<AppModel> {
6    template() {
7      return div({ class: 'app-root'}, [
8        span(`Hello ${this.state.name}! Welcome! `),
9        span(`The answer to the utimate question of life, the universe
             and everything is: ${this.state.answer}`)
10     ]);
11   }
12 }
13 export const app = componentFactory(AppComponent);
```

Primer koda 5.2: Sadržaj fajla `App.ts`

Ukoliko pogledamo aplikaciju u pregledaču (slika 5.2) videćemo da struktura HTML elemenata odgovara strukturi koju vraća funkcija `template()`. Recimo da želimo da odredimo ponašanje našoj aplikaciji, tako da svaki put kada kliknemo na element div povećamo broj iz pozdravne poruke za jedan, a ime promenimo u „Ford". To možemo da uradimo tako što ćemo izmeniti funkciju `template()` na sledeći način:

```
6  template() {
7    return div({ class: 'app-root'}, [
8        span(`Hello ${this.state.name}! Welcome! `),
9        span(`The answer to the utimate question of life, the universe
             and everything is: ${this.state.answer}`)
10   ]).on('click', () => {
11     store.updateState({
12       answer: this.state.answer + 1,
13       name: 'Ford'
14     })
15   });
16 }
```

Primer koda 5.3: Fajl `App.ts` nakon izmenjene funkcionalnosti



Slika 5.3: Aplikacija „Zdravo svete”, nakon izmenjene funkcionalnosti.

Ukoliko otvorimo ponovo našu aplikaciju u pregledaču i kliknemo na nju dva puta, trebalo bi da uočimo da broj koji se prikazuje na kraju poruke više nije 42, već 44, dok je pozdravna poruka ovog puta, umesto Arturu, namenjena Fordu (slika 5.3).

5.3 Struktura Pure-aplikacija

Nakon pravljenja prvog programa u okruženju *Pure*, pogledaćemo detaljnije strukturu i značenje fajlova jedne *Pure*-aplikacije oslanjajući se na primer aplikacije „Zdravo svete”.

5.3.1 Mesto povezivanja

Prvi fajl koji se učitava i izvršava, nakon fajla `index.html` je `index.ts` (Primer 5.4). Zbog toga u njemu pozivamo funkciju za povezivanje (eng. *bootstrap*) aplikacije. Metoda `bootstrap()` prihvata tri argumenta.

1. Referencu na koreni element DOM-drveta aplikacije. To jest, element iz fajla `index.html` koji želimo da zamenimo *Pure* aplikacijom;
2. Konstruktorsku funkciju korene komponente napisane u okruženju *Pure* i
3. Referencu na objekat koji upravlja skladištem (eng. *storage / store*).

Treći argument se prosleđuje funkciji povezivanja kako bi okruženje moglo da osluškuje promene stanja u objektu `store` i reaguje na te promene ponovnim iscrtavanjem elemenata DOM-a.

```
1 import { bootstrap } from "pure-framework/core";
2 import { app } from "./src/App";
3 import { store } from "./src/store";
4
5 bootstrap(document.getElementById('app'), app, store);
```

Primer koda 5.4: Fajl `index.ts`

5.3.2 Komponente

Komponente sadrže logiku i vizuelni opis dela aplikacije. Svaka komponenta u okruženju Pure mora da nasledi apstraktnu klasu `Component<T>`, gde generički argument `<T>` predstavlja tip modela podataka komponente. Konstruktor komponente ne treba pozivati neposredno, već je potrebno napraviri posredničku funkciju (eng. *proxy*) koju možemo napraviti pomoću funkcije `componentFactory` koja se uvozi sa putanje `„pure-framework/core”`. Razlog za ovo, kao i detalji implementacije funkcije `componentFactory` biće pojašnjeni u kasnijim poglavljima.

Klasa `Component<T>` sadrži razne metode, ali ćemo se u ovom delu fokusirati na dve: Metodu `template()` i metodu `render()`.

Metoda `template()` je apstraktna metoda koju mora da implementira svaka komponenta koja nasleđuje klasu `Component<T>`. Ona je zadužena za pravljenje strukture DOM-elemenata komponente i po potpisu funkcije mora da vrati objekat tipa `FunctionalElement`. Tip `FunctionalElement` je interfejs o kome će biti više reči kasnije, ali je za sada dovoljno pomenuti da ovaj interfejs implementiraju sve klase iz kolekcije ugrađenih funkcionalnih elemenata.

Primer jednog funkcionalnog elementa je instanca klase `DivElement`. Instance ovih klasa se nikada ne kreiraju neposredno pozivanjem konstruktora, već isključivo korišćenjem specijalnih *factory*-funkcija[4]. Ovo je važno zbog iskorišćenja optimizacionih mehanizama okruženja, ali i zbog čitljivosti koda.

Važno je napomenuti da i klasa `Component<T>` takođe implementira interfejs `FunctionalElement`, što znači da povratna vrednost metode `template()` takođe može biti i druga komponenta.

[4] *Factory* - eng. Fabrika. U ovom kontekstu: „Šablon Fabrika” - šablon za kreiranje novih objekata.

5.3.3 Funkcionalni elementi

Sve ugrađene *factory*-funkcije za kreiranje funkcionalnih elemenata, koji služe opisu ugrađenih HTML-elemenata, mogu se uvesti sa putanje `,,pure-framework/html''`. Sve ugrađene *factory*-funkcije ovih funkcionalnih elemenata imaju isti potpis. Ove funkcije podrazumevano prihvataju dva argumenta:

1. Objekat atributa.
2. Listu unutrašnjih elemenata. (Lista čvorova-dece)

Važno je napomenuti da ukoliko se prosledi samo jedan argument, podrazumeva se da je u pitanju lista unutrašnjih elemenata, a ne objekat atributa, jer je češći slučaj da HTML element nema definisane atribute, nego što je slučaj da nema podčvorove. Ukoliko element sadrži samo jedan unutrašnji element, možemo i izostaviti pakovanje tog elementa u niz i pustiti da okruženje to uradi umesto nas. Svi ovi izuzeci dodati su u okruženje radi lakše čitljivosti i smanjenja suvišnog koda. U prikazima 5.5 i 5.6 možemo videti kako se prevodi jedna struktura funkcionalnih elemenata u HTML elemente.

```
1  div({ id: 'app-root',  }, [
2    span('First span'),
3    span('Second span'),
4    div({ class: 'inner-div'}, [
5      span([
6        span('This is left from'),
7        a({ href:'https://google.
             com' }, ['Link']),
8        span(', and this one is on
              the right of it.'),
9      ])
10   ])
11 ]);
```

Primer koda 5.5: Funkcionalni element

```
1  <div id="app-root">
2    <span>First span</span>
3    <span>Second span</span>
4    <div class="inner-div">
5      <span>
6        <span>This is left from</
             span>
7        <a href="https://google.
             com"> Link </a>
8        <span>, and this one is on
              the right of it.</span
             >
9      </span>
10   </div>
11 </div>
```

Primer koda 5.6: Rezultujući HTML

5.3.4 Model i skladište podataka

Modeli u okruženju Pure ne sadrže nikakvu logiku i pišu se pomoću interfejsa u *TypeScript*-u, tako da postoje samo u fazi razvoja. U fazi izvršavanja, ovi fajlovi nisu deo izvornog koda, zato što se uklanjaju u postupku prevođenja. Njihova svrha je samo da nam obezbede striktne tipove pri navođenju podrazumevanih vrednosti i spreče eventualne logičke greške u fazi izvršavanja. Model aplikacije „Zdravo svete" može se videti na primeru 5.7.

```
1 export interface AppModel {
2   name: string;
3   answer: number;
4 }
```

Primer koda 5.7: Fajl `src/model.ts`

Skladište koje se koristi u okviru okruženja Pure zasnovano je na mehanizmu *BehaviorSubject*-a[5] iz biblioteke *RxJs*. Kako bismo napravili jednu instacu skladišta za našu aplikaciju, potrebno je da pozovemo konstruktor `Store<T>` koji moramo uvezemo sa putanje `,,pure-framework/core''`. Konstruktor `Store<T>` prima jedan generički argument tipa modela, i jedan standardni argument koji predstavlja objekat podrazumevanog stanja koji odgovara po svojoj strukturi prosleđenom generičkom argumentu (Primer 5.8).

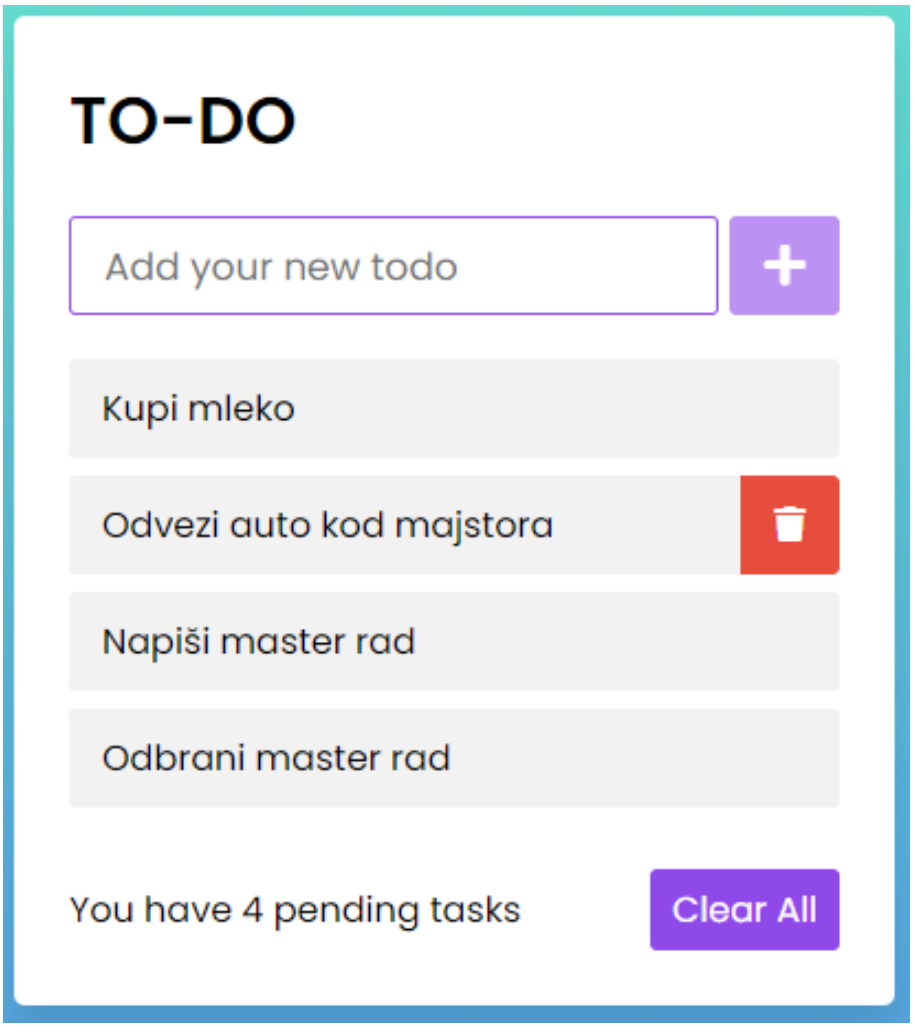
```
1 import { Store } from "pure-framework/core";
2 import { AppModel } from "./model";
3
4 export const store = new Store<AppModel>({
5     name: 'Arthur',
6     answer: 42
7 })
```

Primer koda 5.8: Fajl `src/store.ts`

[5] *BehaviorSubject* - eng. Subjekt ponašanja

5.4 Aplikacija „Menadžer Zadataka”

Pogledaćemo sada malo kompleksniji primer. U pitanju je aplikacija za upravljanje dnevnim obavezama[6]. Online verzija ove aplikacije nalazi se na stranici `https://pure-framework-todo-demo.netlify.app/` (Slika 5.4). Izvorni kod aplikacije nalazi se na adresi:
`https://github.com/maleksandar/pure-framework-todo-app`



Slika 5.4: Online verzija TO-DO aplikacije

[6] *eng. TO-DO* lista.

5.4.1 Struktura aplikacije

Pogledajmo sada isečak koda koji odgovara korenoj komponenti `ToDoApp` koja se nalazi u fajlu `src/ToDoApp.ts`:

```
1   import { ... }; // Lista zavisnosti skracena zbog citljivosti
    ...
7   class ToDoApp extends Component<ToDoState> {
    ...
19    template() {
20      return div({ class: 'wrapper' }, [
21        this.headerSegment(),
22        this.todoSegment(),
23        this.footerSegment(),
24      ]);
25    }
    ...
46    private todoSegment() {
47      return ul({ class: 'todoList' },[
48        ...this.state.todoList.map((item) =>
49          li([
50            item,
51            span({ class: 'icon' }, [
52              italic({ class: 'fas fa-trash' }, [])
53            ]).on('click', () => {
54              let index = this.state.todoList.findIndex(x => x == item);
55              this.state.todoList.splice(index, 1);
56              store.updateState(this.state);
57            })
58          ]))
59      ]);
60    }
    ...
109 }
110
111 export const todoApp = componentFactory<ToDoApp, ToDoState>(ToDoApp);
```

Primer koda 5.9: Fajl `ToDoApp.ts`

Ukoliko pogledamo strukturu liste potomaka u metodi `template()` (Primer 5.9) videćemo da ona upravo opisuje tri vizualna segmenta aplikacije 5.4:

1. Zaglavlje sa naslovom, poljem za unos novog zadatka i dugmetom za potvrdu unosa (Funkcija `headerSegment()`);

2. Segment sa listom aktuelnih zadatka (Funkcija `todoSegment()`) i
3. Podnožje sa prikazom zbirnih informacija i dugmetom za brisanje liste aktuelnih zadataka (Funkcija `footerSegment()`).

U primeru koda 5.9 prikazan je kod funkcije koja opisuje deo aplikacije sa listom aktuelnih zadataka. Funkcija `todoSegment()` vraća `UlElement` funckcionalni element (povratna vrednost *factory*-funkcije `ul()`), koji odgovara HTML-elementu `<ul>`). U redu 48 vidimo poziv operatora *spread (eng. raširi)* nad nizom koji se dobija kao rezultat mapiranja niza `this.state.todoList` u niz elemenata `li`. Svaki od elemenata `li` takođe sadrži dva deteta-čvora, od kojih jedan predstavlja nisku koja sadrži opis zadatka i dugme za uklanjanje zadatka.

Funkcija za uklanjanje pojedinačnih zadatka opisana je kodom koji počinje u redu 54, primera 5.9. Primetimo bitan detalj u redu 56, u kojem pozivamo metodu nad objektom `store` sa novim, izmenjenim stanjem. Ovaj poziv je neophodan kako bismo korenoj komponenti signalizirali promenu stanja aplikacije, i naterali okruženje da ponovo izvrši iscrtavanje.

Ova veza između komponente `todoApp` i objekta `store` ostvarena je pozivanjem metode `bootstrap()` u fajlu `index.ts` (Primer 5.4).

Pogledajmo sada šta sadrži objekat `store`, koji uvozimo iz fajla `src/store.ts` (Primer 5.10).

```
1  import { Store } from "pure-framework/core";
2  import { ToDoState } from "./model";
3
4  export const store = new Store<ToDoState>({
5    todoList: [
6        'Kupi mleko',
7        'Odvezi auto kod majstora',
8        'Napisi master rad',
9        'Odbrani master rad',
10   ]
11 });
```

Primer koda 5.10: Fajl `src/store.ts`

Na osnovu prikaza 5.10 vidimo da je objekat `store` instanca klase `Store` koja je definisana u okviru paketa `pure-framework`. Konstruktor ove klase prima generički argument (`ToDoState`) koji opisuje model podataka naše aplikacije (model podataka korene komponente) i inicijalizujući objekat, koji se koristi kao podrazumevano stanje komponente ili u ovom slučaju cele aplikacije.

Model podataka aplikacije opisan je *TypeScript*-interfejsom koji je definisan u fajlu `src/model.ts` (Primer 5.11)

```
1 export interface ToDoState {
2   todoList: string[];
3 }
```

Primer koda 5.11: Fajl `src/model.ts`

Glava 6

Implementacija okruženja Pure

U ovom poglavlju upustićemo se u detalje implementacije okruženja *Pure* i objasniti neke odluke donete tokom dizajniranja samog okruženja.

6.1 Korišćeni alati, tehnologije i biblioteke

Za rešavanje većine problema opisanih u poglavlju 3 postoje različiti alati i biblioteke. Kako bi fokus okruženja *Pure* trebalo da bude na do sada nerešenim (ili nedovoljno adekvatno rešenim) problemima, potrebno je odabrati i uvezati ove postojeće alate na pravi način. U ovoj sekciji ćemo opisati glavne alate i biblioteke koje predstavljaju deo okruženja *Pure*, ili su korišćene tokom razvoja okruženja.

6.1.1 Typescript

TypeScript je strogo tipizirani programski jezik izgrađen oko jezika *JavaScript*. Prednost korišćenja strogih tipova u pisanju aplikacija leži u mogućnosti programskih alata da veliki opseg grešaka otkriju u fazi razvoja. Primera radi, ukoliko neka funkcija kao ulazni argument očekuje nisku, a mi nad tim argumentom pozivamo metodu koja nije definisana na prototipu[1] niske, kompilator će nam izbaciti grešku i upozoriti nas da to (verovatno) nije metoda koju smo želeli da pozovemo [17].

Pošto *TypeScript* predstavlja suštinski samo nadskup jezika *JavaScript* (korišćenje tipova je opciono), možemo jednostavno balansirati između korišćenja striktnih tipova i sigurnosti sa jedne strane i čitljivosti koda sa druge, tako da

[1]Sistem nasleđivanja u jeziku *JavaScript* je baziran na prototipovima [17].

je cena uvođenja ove tehnologije u okruženje veoma niska, a dobici mogu biti značajni.

6.1.2 Parcel

Parcel je popularan upakivač (eng. *bundler*) za veb aplikacije. Upakivač u razvojnim okruženjima ima dva zaduženja:

1. Pravljenje *produkcionog JavaScript* fajla sa minifikovanim kodom [15].
2. Povezivanje koda i veb pregledača tokom pisanja koda.

„Produkcioni JavaScript fajl" je fajl koji sadrži celokupnu logiku aplikacije, ali je za potrebe brzog preuzimanja i učitavanja od strane pregledača minifikovan i preveden na nižu verziju standarda *ECMAScript* [16] (zbog starijih verzija pregledača). Ukratko, sve beline iz koda su uklonjene, a promenljive preimenovane u kraća simbolička imena. Napredni konstrukti jezika prevode se na semantičke ekvivalente iz starije verzije *ECMAScript*-a.

Glavna prednost upakivača *Parcel* je u minimalnoj neophodnoj konfiguraciji. Konkretan razlog zbog kog je *Parcel* odabran kao upakivač, je taj što *Parcel* automatski prevodi *TypeScript* kod na *JavaScript* bez ikakvog dodatnog podešavanja, pa zbog toga možemo u okviru fajla `index.html` neposredno da zahtevamo *TypeScript* fajl (Primer 6.1).

```
<script src="index.ts" type="module"></script>
```

Primer koda 6.1: Uključivanje *TypeScript* fajla neposredno u `index.html`

U ovom konkretnom primeru (Primer 6.1), *Parcel* će u pozadini prevesti *TypeScript* kod na *JavaScript* i zameniti referencu ka *TypeScript*-fajlu minifikovanim rezultujućim *JavaScript*-fajlom u momentu upakivanja koda.

6.1.3 RxJs

RxJs je biblioteka koja sadrži generičku implementaciju obrazca *Observer*[2] za *JavaScript* [6] [4]. Deo je većeg skupa alata otvorenog koda pod nazivom *ReactiveX* koji služe reaktivnom programiranju [3].

Reaktivno programiranje podrazumeva programiranje vođeno događajima. Popularna implementacija šablona *Redux* koja se koristi za *Angular* aplikacije pod nazivom *NgRx* je implementirana upravo nad ovom bibliotekom [2] [25]. Okruženje *Pure* koristi biblioteku *RxJs* iz istih razloga, ali u značajno pojednostavljenoj varijanti.

6.1.4 Ostali alati i biblioteke

Pored glavnih alata opisanih u ovom poglavlju, korišćeni su i alati otvorenog koda za manje i jednostavnije zadatke. Neki od njih su:

1. `jest` - Okruženje za testiranje *JavaScript* koda.
2. `jsdom` - Implementacija pregledača i DOM-a u redukovanoj konzolnoj verziji zarad lakšeg automatskog testiranja.
3. `object-hash` - Implementacija različitih heš algoritama za heširanje *JavaScript* objekata, zarad brzog poređenja objekata po vrednosti.
4. `rfdc` - Implementacija algoritma za *duboko kloniranje*[3] *JavaScript* objekata.

6.2 Povezivanje

Kao što je već pomenuto u sekciji 5.3.1, `index.ts` je prvi fajl koji biva učitan nakon fajla `index.html`. Jedina funkcija koju pozivamo u ovom fajlu (u do sada prikazanim primerima) je funkcija za povezivanje aplikacije: `bootstrap()`. Njena implementacija nalazi se u fajlu `core/bootstrap.ts` i može se videti u primeru 6.2. Funkcija `bootstrap()` je generička funkcija koja prihvata jedan argument, čiji tip odgovara tipu korenog modela podataka, i tri standardna argumenta. Standardni argumenti su:

1. Referenca na element DOM-a;

[2] *Observer* - eng. Posmatrač.

[3] Pri dubokom kloniranju reference ka pod-objektima se ne dele između originalnog i kloniranog objekta, već se svi pod-objekti takođe kloniraju.

2. Konstruktorska funkcija komponente i
3. Referenca na objekat skladišta.

Ovi argumenti su već opisani u sekciji 5.3.1, pa njihovo značenje nećemo ovde ponovo navoditi.

```
1  import { Component } from "./component";
2  import { Store } from "./store";
3
4  export function bootstrap<AppModel>(
5      domRoot: HTMLElement,
6      rootComponentFactory: (state: () => AppModel) => Component<
           AppModel>,
7      store: Store<AppModel>,
8  ): Component<AppModel> {
9
10     const appRoot = rootComponentFactory(() => store.state);
11     store.state$.subscribe(state => {
12         while (domRoot.firstChild) {
13             domRoot.removeChild(domRoot.lastChild);
14         }
15         domRoot.appendChild(appRoot.render());
16     });
17
18     return appRoot;
19 }
```

Primer koda 6.2: Fajl `core/bootstrap.ts`

Ukoliko bliže analiziramo prikazani kod, primetićemo da funkcija `bootstrap()` ima 3 zaduženja:

1. Instanciranje korene komponente pozivom prosleđene *factory*[4]-funkcije (Linja 10) [6].
2. Osluškivanje promene stanja skladišta (Linija 11).
3. Registrovanje anonimne funkcije za ponovno iscrtavanje elemenata DOM-a (Linije 11-16).

Iako će pojedinačni detalji komponenata, fabrike komponenata, skladišta i obrade događaja biti objašnjeni u kasnijim poglavljima, ovde ćemo navesti neke važnije aspekte ovih elemenata, kako bismo jasnije razumeli prikazani kod.

[4] *Factory* - eng. Fabrika. U ovom konteksu „obrazac Fabrika" [6].

Primetimo da drugi argument koji prihvata funkcija `bootsrap()`, u samom kodu nazivamo `rootComponentFactory`. Razlog je u tome što prosleđena funkcija nije samo jednostavni konstruktor, već ima dodatnu funkcionalnost u domenu optimizacije. U liniji 11 vidimo poziv metode `subscribe()` nad objektom `store.state$`. Ovo je metoda koja je definisana u okviru biblioteke *RxJs* nad tipom `Observable<T>.` Više detalja o ovom i drugim tipovima iz biblioteke *RxJs* biće u odeljku 6.6.

6.3 Funkcionalni Elementi

Tip `FunctionalElement` predstavlja interfejs najopštijeg nivoa u hijerarhiji strukturnih elemenata okruženja *Pure*[5], koji sadrži minimalan skup metoda koje se podrazumevano mogu pozvati nad bilo kojim elementom. U kodu je definisan *TypeScript* interfejsom na putanji `core/functionalElement.ts` (Primer 6.3).

```
1 export interface FunctionalElement {
2   render: () => HTMLElement;
3   domElement: HTMLElement;
4   children?: FunctionalElement[];
5   parentDomElement: HTMLElement;
6 }
```

Primer koda 6.3: Fajl `core/functionalElement.ts`

Interfejs `FunctionalElement` deklariše sledeće metode i polja:

1. Metodu `render()` koja se poziva pri iscrtavanju DOM-elemenata;
2. Referencu na DOM-element koji napravljen pozivanjem metode `render()`;
3. Listu potomaka i
4. Referencu na „roditelja" DOM-elementa (`null` u slučaju korene komponente).

Primetimo da prikazani interfejs (Primer 6.3) pored metode (`render()`) sadrži i polja. Ovo je specifično za jezik *TypeScript* u kojem interfejsi mogu definisati i polja, a ne samo metode[6].

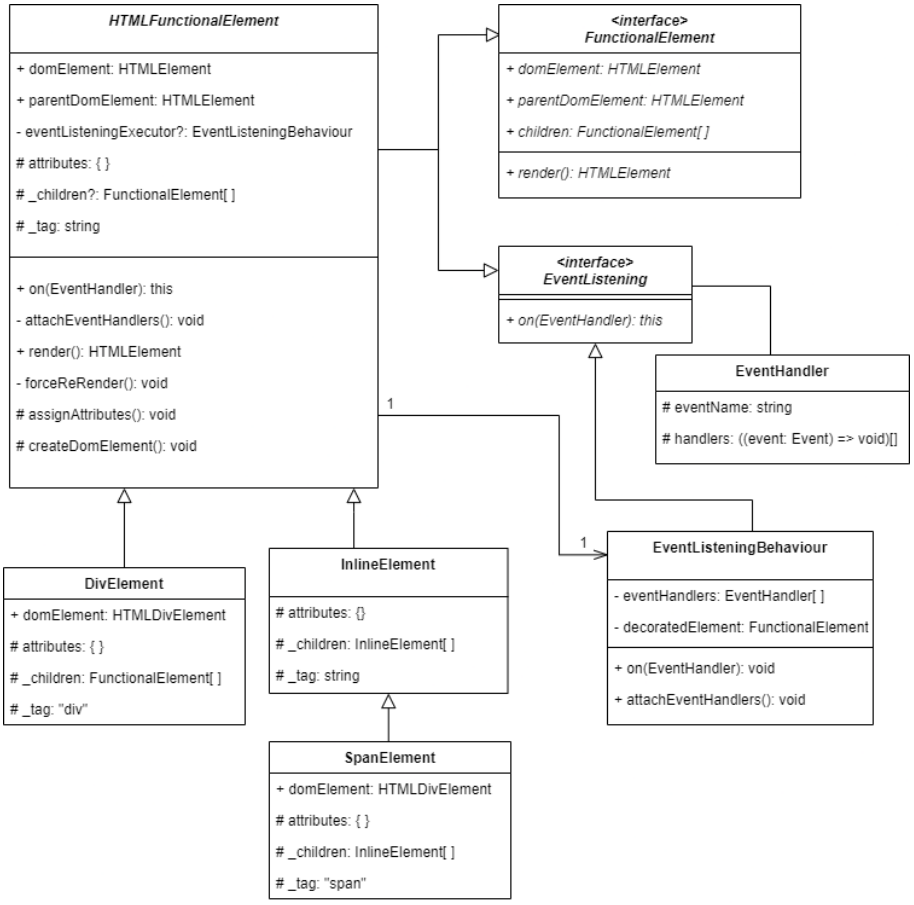
[5] Elementi za opis DOM-strukture.

[6] Pošto neposredno pristupanje poljima instance nije redak slučaj u *JavaScript*-u, inženjeri *TypeScript*-a su odlučili da zvanično tako nešto podrže i u svojoj sintaksi.

6.3.1 Ugrađeni Funkcionalni Elementi

Kako bismo izgradili nove komponente koristeći šablon kompozicije (eng. *Composite pattern*), moramo se osloniti na neke postojeće, to jest, elemente ugrađene u okruženje. Elementi koji su ugrađeni u okruženje *Pure* su elementi koji predstavljaju pandan postojećim HTML elementima (`div`, `h1`, `span`, ...).

U nastavku poglavlja prikazaćemo implementaciju funkcionalnog elementa `div` kao i dijagram klasa koji opisuje hijerarhiju i odnose elementa `div` sa ostalim elementima. Implementacija ostalih funkcionalnih elemenata može se naći na repozitorijumu okruženja *Pure*.



Slika 6.1: Dijagram klasa za klasu `HTMLFunctionalElement`

Na osnovu dijagrama (slika 6.1) vidimo da `DivElement` nasleđuje klasu `HTMLFunctionalElement`. Ukoliko pogledamo kod klase `DivElement` (Primer 6.4) primetićemo da klasa `DivElement` ne dodaje nikakvo ponašanje u odnosu na klasu `HTMLFunctionalElement`, osim što fiksira argument imena elementa (`_tag`).

```
import { FunctionalElement, HTMLFunctionalElement } from '/core';
export class DivElement extends HTMLFunctionalElement {
    constructor(protected attributes: {}, protected _children:
        FunctionalElement[]) {
         super(attributes, _children, 'div')
    }
}
```

Primer koda 6.4: Fajl `html/block/elements/divElement.ts`

Kao što se da zaključiti iz njenog naziva, `HTMLFunctionalElement` je klasa koja definiše ponašanje svih ugrađenih elemenata HTML-a. Pogledajmo sada detalje implementacije klase `HTMLFunctionalElement`:

```
import { EventListening } from "./eventListening";
import { EventListeningBehaviour } from "./eventListeningBehaviour";
import { FunctionalElement } from "./functionalElement";

export abstract class HTMLFunctionalElement implements
    FunctionalElement, EventListening {
  public domElement: HTMLElement;
  public parentDomElement: HTMLElement;
  private _eventListeningExecutor: EventListeningBehaviour = null;

  constructor(protected attributes: {}, protected _children:
      FunctionalElement[], protected _tag: string) {
    if (arguments.length === 2) {
      // in case attributes object is provided as a first argument
      this.attributes = arguments[0];
      this._children = arguments[1];
    }
    else if (arguments.length === 1) {
      // in case we are provided with only one argument - we asume it
          is the array of children or a single child
      this._children = arguments[0];
    }
    this._eventListeningExecutor = new EventListeningBehaviour(this);
  }

  public on(event: string, ...handlers: ((event: Event) => void)[]) {
    this._eventListeningExecutor.on(event, ...handlers);
    return this;
  }
```

```
27
28    public get children(): (FunctionalElement)[] {
29      return this._children;
30    }
31
32    public render(): HTMLElement {
33      this.createDomElement();
34      this.assignAttributes();
35      this.attachEventHandlers();
36      return this.domElement;
37    }
38
39    public forceReRender() {
40      let domElementToReplace = this.domElement;
41      this.parentDomElement.replaceChild(this.render(),
            domElementToReplace);
42    }
43
44    protected assignAttributes(): void {
45      Object.keys(this.attributes).forEach(attribute => {
46        if (this.attributes[attribute]) {
47          this.domElement.setAttribute(attribute, this.attributes[
                attribute]);
48        }
49      });
50    }
51
52    protected createDomElement(): void {
53      this.domElement = document.createElement(this._tag);
54
55      this.children.forEach(child => {
56        child.parentDomElement = this.domElement;
57        this.domElement.appendChild(child.render());
58      });
59    }
60
61    private attachEventHandlers() {
62      this._eventListeningExecutor.attachEventHandlers();
63    }
64  }
```

Primer koda 6.5: Fajl `core/htmlFunctionalElement.ts`

Metoda `render()` klase `HTMLFunctionalElement` interno poziva tri privatne metode:

1. Metodu za pravljenje elementa DOM-a (`createDomElement()`);
2. Metodu za dodeljivanje potencijalno prosleđenih atributa napravljenom DOM-elementu (`assignAttributes()`) i
3. Metodu za registraciju osluškivača događaja (`attachEventHandlers()`).

U privatnoj metodi `createDomElement()`, nakon samog poziva za kreiranje DOM-elemenata se proverava da li je funkcionalnom elementu prosleđena lista potomaka. Ukoliko jeste, metoda iterira kroz tu listu i rekurzivno za svako dete poziva metodu `render()` i na taj način se formira celo drvo DOM-elemenata (Primer 6.5, Linije 54-63).

Privatna metoda `assignAttributes()` prolazi kroz listu atributa koje imaju vrednost *truthy*[7] i dodeljuje prosleđene atribute DOM-elementu.

Mehanizam registrovanja osluškivača događaja i samo upravljanje događajima biće obrađeno u sekciji 6.6.

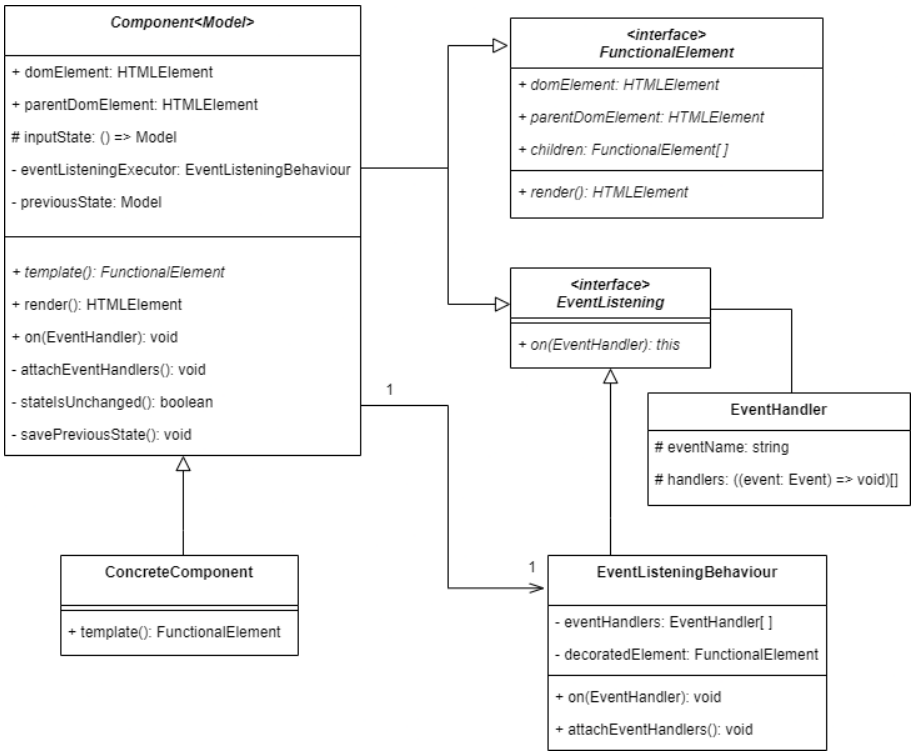
6.4 Komponente

Komponente u okruženju *Pure* predstavljaju funkcionalne elemente koji se mogu dinamički menjati u odnosu na promenu stanja skladišta. Sve komponente moraju da naslede apstraktnu klasu `Component<T>`. Ova apstraktna klasa ima jednu apstraktnu metodu `template()` koja se mora definisati u potklasi.

Metoda `template()` mora da vrati tip koji implementira interfejs `FunctionalElement`. To može biti druga komponenenata ili neki od ugrađenih funkcionalnih elemenata.

Pošto klasa `Component<T>` implementira interfejs `FunctionalElement`, mora da implementira i metodu `render()`. Metoda `render()` najpre proverava da li je stanje promenjeno u odnosu na prethodno i ukoliko nije, vraća DOM-element koji je napravljen u prethodnom pozivu. Ukoliko je stanje promenjeno, trenutno stanje se čuva (kako bi moglo da se uporedi sa budućim), a nad objektom koji vraća metoda `template()` se poziva metoda `render()`. Rezultat tog poziva se vraća kao izlazna vrednost metode `render()`.
Pogledajmo sada implementaciju klase `Component<T>`.

[7]Skup vrednosti u *JavaScript*-u koje se konverzijom prevode na *boolean* vrednost `true` [18].



Slika 6.2: Dijagram klasa za klasu `Component<Model>`

```
1  import { FunctionalElement } from "./functionalElement";
2  import { EventListeningBehaviour } from "./eventListeningBehaviour";
3  import { areEqual, cloneDeep } from "../utils";
4  import { EventListening } from "./eventListening";
5
6  export abstract class Component<ModelType>  implements
       FunctionalElement, EventListening {
7      public domElement: HTMLElement | Text = null;
8      public parentDomElement: HTMLElement = null;
9
10     protected inputState: () => ModelType = () => null;
11
12     private _eventListeningExecutor: EventListeningBehaviour = null;
13     private previousState: ModelType = null;
14
15     constructor(inputState: () => ModelType) {
16         this._eventListeningExecutor = new EventListeningBehaviour(
               this);
17         this.inputState = inputState;
18     }
```

```
19
20      abstract template(): FunctionalElement;
21
22      get state() {
23          return this.inputState();
24      }
25
26      get children() {
27          return this.template().children
28      }
29
30      public render() {
31          if (this.stateIsUnchanged() && this.domElement) {
32              return this.domElement;
33          }
34
35          this.savePreviousState();
36          this.domElement = this.template().render();
37          this.attachEventHandlers();
38
39          return this.domElement;
40      }
41
42      public on(event: keyof HTMLElementEventMap, ...handlers: ((event:
            Event) => void)[]) {
43          this._eventListeningExecutor.on(event, ...handlers);
44          return this;
45      }
46
47      private attachEventHandlers() {
48          this._eventListeningExecutor.attachEventHandlers();
49      }
50
51      private stateIsUnchanged() {
52          return areEqual(this.inputState(), this.previousState);
53      }
54
55      private savePreviousState() {
56          this.previousState = cloneDeep(this.inputState());
57      }
58  };
```

Primer koda 6.6: Fajl `core/component.ts`

6.5 Obrada događaja

Obratimo pažnju na pojedine elemente na dijagramu klasa koji opisuje okruženje klase `Component<Model>` i dijagramu klasa koji opisuje okruženje klase `HTMLFunctionalElement`. Videćemo da pored osnovnog interfejsa svih elemenata (`FunctionalElement`) obe klase implementiraju interfejs `EventListening` i sadrže instancu klase koja neposredno implementira ovaj interfejs (`EventListeningBehaviour`). Razlog za ovo je potreba za razdvajanjem interfejsa (eng. *Interface Segregation*) i lakšim razvojem koda kroz princip *„kompozicija ispred nasleđivanja"* (eng. *Composition over Inheritance*).

Korišćenjem (pojednostavljenog) obrazca *Dekorator* [6] razdvajamo implementaciju za osluškivanje i reagovanje na korisničke događaje.

Pogledajmo detalje ovog interfejsa, kao i konkretnu implementaciju (ista implementacija interfejsa se koristi i za `HTMLFunctionalElement` i za `Component<Model>`).

```
1 export interface EventListening {
2   on: (event: keyof HTMLElementEventMap, ...handlers: ((event: Event)
        => void)[]) => this,
3 }
```

Primer koda 6.7: Fajl `core/eventListening.ts`

```
1  import { EventListening } from "./eventListening";
2  import { FunctionalElement } from "./functionalElement";
3
4  export class EventListeningBehaviour implements EventListening {
5    private eventHandlers: { event: keyof HTMLElementEventMap; handlers:
         ((event: any) => any)[]; }[] = [];
6    constructor(private decoratedElement: FunctionalElement) { }
7
8    public on(event: keyof HTMLElementEventMap, ...handlers: ((event:
       Event) => void)[]) {
9      this.eventHandlers.push({event, handlers});
10     return this;
11   };
12
13   public attachEventHandlers(): void {
14     if (this.eventHandlers) {
15         this.eventHandlers.forEach(event => {
16             event.handlers.forEach(handler => {
```

```
17                this.decoratedElement.domElement.addEventListener(event.
                      event, handler);
18              });
19          });
20      }
21    }
22  }
```

Primer koda 6.8: Fajl `core/eventListeningBehaviour.ts`

Obratimo sada pažnju na isečak koda iz implementacije klase `Component<T>` koji smo videli u primeru 6.6:

```
1   import { ... } from "...";
      ...
15      constructor(inputState: () => ModelType) {
16          this._eventListeningExecutor = new EventListeningBehaviour(
                this);
17          this.inputState = inputState;
18      }
      ...
42      public on(event: keyof HTMLElementEventMap, ...handlers: ((event:
            Event) => void)[]) {
43          this._eventListeningExecutor.on(event, ...handlers);
44          return this;
45      }
46
47      private attachEventHandlers() {
48          this._eventListeningExecutor.attachEventHandlers();
49      }
      ...
58  };
```

Primer koda 6.9: Isečak koda iz fajla `core/component.ts`

U primerima 6.8 i 6.9 vidimo mehanizam delegacije ponašanja pri implementiranju interfejsa. Naime, pošto suštinski i komponente i ugrađeni funkcionalni elementi reaguju na događaje na gotovo isti način, ima smisla to ponašanje definisati u okviru posebne klase koju ne mora nasleđivati ni jedna od ove dve klase. Na taj način činimo strukturu koda prilagodljivom budućim promenama u hijerarhiji klasa.

Klasa `EventListeningBehaviour` ima za cilj da dekoriše funkcionalni element metodama `on()` i `attachEventHandlers()`.

6.6 Promena stanja

Pogledajmo sada malo detaljnije kako *Pure*-aplikacija implementira mehanizam skladišta, analizirajući kod iz primera 6.10:

```
1  import { BehaviorSubject } from "rxjs";
2
3  export class Store<Model> {
4      private _stateSubject: BehaviorSubject<Model>;
5
6      constructor(defaultState: Model) {
7          this._stateSubject = new BehaviorSubject(defaultState);
8      }
9
10     updateState(newState: Partial<Model>) {
11         this._stateSubject.next({...this._stateSubject.getValue(), ...
               newState });
12     }
13
14     get state() {
15         return this._stateSubject.getValue();
16     }
17
18     get state$() {
19         return this._stateSubject.asObservable();
20     }
21 }
```

Primer koda 6.10: Fajl `core/store.ts`

Skladište u razvojnom okruženju *Pure* se zasniva na tipu `BehaviorSubject<T>` iz biblioteke *RxJs*. `BehaviorSubject<T>` je klasa koja objedinjuje posmatrača i subjekta posmatranja u obrazcu *Observer*[8] [6]. Drugim rečima, nad objektom tipa `BehaviorSubject<T>` možete da registruje funkciju za obradu događaja pri promeni stanja, ali i pozvati metodu za promenu stanja.

```
1  behaviorSubject.subscribe(newState => { /* handling new state */});
```

Primer koda 6.11: Primer registrovanja osluškivača stanja

```
1  behaviorSubject.next({...oldState, changedProperty: 'new value'});
```

Primer koda 6.12: Primer emitovanja novog stanja

[8] *Observer* - eng. Posmatrač

Ovaj mehanizam omogućava nam implementaciju centralizovanog sistema za skladištenje podataka. Ako pogledamo kod konkretne implementacije skladišta 6.10, videćemo da klasa ima jednu javnu metodu i dva javna svojstva:

1. Svojstvo nad kojim vršimo registraciju osluškivača (`state$`);
2. Svojstvo za vraćanje trenutnog stanja (`get state()`) i
3. Metodu za promenu stanja (`updateState()`).

Važno je napomenuti da će se svakim pozivom metode `updateState()`, interno stanje skladišta promeniti. To jest, nakon svakog poziva metode `updateState()`, pozivanje svojstva `get state()` vratiće novi objekat. To nam garantuje da osnovni princip šablona *Flux* neće biti narušen.

6.7 Memoizacija

Već smo napomenuli da se komponente nikada ne instanciraju neposredno preko svojih konstruktora, već isključivo preko *factory*-funkcija koje možemo napraviti koristeći ugrađenu funkciju `componentFactory` koju možemo uvesti sa putanje `„pure-framework/core”` (Primer 6.13). Razlog za to leži u mehanizmu optimizacije implementiranom u okruženju *Pure*, koji se bazira na principu memoizacije čistih funkcija [21].

Memoizacija je ukratko mehanizam koji podrazumeva keširanje rezultata izlaznih vrednosti čistih funkcija, po ključu koji se dobija od ulaznih parametara funkcije. Za implementaciju ovog mehanizma, neophodno je da funkcije budu „čiste”, to jest, da za iste ulazne parametre uvek proizvode isti izlazni rezultat.

Pogledajmo konkretnu implementaciju ovog mehanizma u funkciji `componentFactory` u primeru 6.13[9]:

```
1  import oh from 'object-hash';
2  import { Component } from './component';
3  type FuncCompConstr<ModelType> = { new(state: () => ModelType):
       Component<ModelType>; }
4
5  const dictionary = Object.create(null);
6
7  export function componentFactory<ConcreteComponent extends Component<
       ModelType>, ModelType>(constrFunc: FuncCompConstr<ModelType>)
8      : (state: () => ModelType, id?: number | string) =>
           ConcreteComponent {
9      return (state: () => ModelType, id: number | string = 0) => {
10         const stateHash = oh.MD5(state());
11         const key = `${stateHash}_${id}`
12         if (!dictionary[constrFunc.name]) {
13             dictionary[constrFunc.name] = Object.create(null)
14         }
15         if (!dictionary[constrFunc.name][key]) {
16             dictionary[constrFunc.name][key] = new constrFunc(state);
17         }
18         return dictionary[constrFunc.name][key] as ConcreteComponent;
19     };
20 };
```

Primer koda 6.13: Fajl `core/componentFactory.ts`

[9]Imena tipova i promenljivih su skraćeni, radi preglednijeg preloma teksta.

Funkcija `componentFactory()` je funkcija višeg reda. To znači da je povratna vrednost ove funkcije druga funkcija. Vraćena funkcija ima potpis koji prihvata dva argumenta:

1. Funkciju za izračunavanje trenutnog stanja (`state: () => ModelType`) i
2. Opcioni identifikator komponente (`id?: number | string`).

U liniji 5 primera 6.13 vidimo inicijalizaciju jednog rečnika (praznog objekta). Ovaj rečnik služi kao keš mehanizam za korisnički definisane komponente. Taj mehanizam funkcioniše na sledeći način:

1. Hešira trenutno stanje komponente koje se dobija pozivanjem funkcije `state()` ;
2. Heš kodu se dodaje opcioni identifikator komponente i tako spojeni čine ključ rečnika `dictionary` ;
3. Proverava se da li u rečniku postoji objekat sa tim ključem;
4. Ukoliko ne postoji pravi se nova instanca komponente pozivanjem prosleđenog konstruktora i čuva se u rečniku pod izračunatim ključem. Novonapravljena instanca se vraća kao rezultat funkcije;
5. Ukoliko postoji vraća se instanca iz rečnika.

Opcioni identifikator komponente (`id`) služi za (retke) slučajeve kada dve iste komponente koristimo na dva različita mesta u aplikaciji sa potpuno istim stanjem. U slučaju nedodavanja ovog identifikatora, ovaj mehanizam bi napravio samo jednu instancu komponenente umesto dve koliko je potrebno.

Glava 7

Diskusija

U ovom poglavlju ćemo analizirati ispunjenost postavljenih zahteva i razmotriti moguća poboljšanja datog rešenja.

7.1 Analiza ispunjenosti zahteva

Osvrnimo se još jednom na listu zahteva iznetu u sekciji 4.1:

Zahtev 1, koji se odnosi na (ne)korišćenje specijalizovanih sintaksi koje se prevode na HTML, ispunjen je korišćenjem koncepta funkcionalnih elemenata. Umesto opisa HTML strukture korisnik pozivom funkcija (fabričkih konstruktora funkcionalnih elemenata) opisuje strukturu DOM-drveta i na taj način koristi sve benefite statičke provere ispravnosti, sintaksnog obeležavanja, lakog refaktorisanja i svega ostalog što jedan moderan programski jezik podrazumevano donosi.

Zahtev 2, koji se odnosi na implementaciju šablona *Flux* ispunjen je implementacijom `Store<Model>` mehanizma za skladištenje podataka. Podaci se (podrazumevano) kreću u jednom smeru kroz aplikaciju koristeći šablon *Posmatrač* implementiran u biblioteciji *RxJs*.

Zahtev 3, koji se odnosi na uočavanje grešaka u fazi razvoja ostvaren je statičkim tipovima i korišćenjem jezika *TypeScript*.

Objavljivanjem paketa u javni *npm*-registar, ispunjen je i zahtev 4.

Zahtev 5, koji se odnosi na otvorenost za modifikaciju, ostvaren je objavljivanjem repozitorijuma pod licencom MIT [14].

Zahtev 6, koji se odnosi na lako čitljivu strukturu ostvaren je minimalnim kodom neophodnim za funkcionisanje aplikacije, na sličan način na koji je to rešeno i u okruženju *React*. Jedina obavezna metoda u okviru implementacije jedne

komponente je metoda `template()`, koja definiše HTML strukturu komponente. Sve ostale metode su opcione, pa korisnik ima široku kontrolu nad organizacijom svog koda.

7.2 Prostor za unapređenja

Okruženje *Pure* ima za cilj da adresira neke od problema koji su prisutni u popularnim okruženjima za razvoj klijentskih veb aplikacija. Kao takavo, više predstavlja dokaz koncepta (eng. *Proof of Concept*), nego rešenje spremno za komercijalnu upotrebu. Ono predstavlja osnovu jednog kompletnijeg i robustnijeg okruženja, koje će moći da se razvije na temeljima trenutnog rešenja.

Kao potencijalna poboljšanja trenutnog rešenja, autor ovog rada vidi u unapređenju strukture koda, dodavanju funkcionalnosti i podeli rešenja na manje funkcionalne celine. U ovoj sekciji ćemo obrazložiti neke od tih ideja.

7.2.1 Struktura koda

Kako bi trenutno rešenje moglo dalje da se razvija i napreduje u okviru zajednice otvorenog koda, neophodno je najpre detaljno dokumentovati i opisati različite modele upotrebe trenutno postojećeg rešenja. Jedan deo tog posla sastoji se od dodavanja *JSDoc* komentara na sve javne metode koje se pozivaju iz korisničkog okruženja u okviru razvoja jedne *Pure*-aplikacije [13]. Drugi deo odnosi se na refaktorisanje trenutnog rešenja i prelazak sa modela nasleđivanja na model kompozicije klasa i ekstenzjiju specifičnih ograničenja za ugrađene funkcionalne elemente, kako bi iskustvo pisanja koda bilo što lakše, a potencijalne greške lako uočljive.

7.2.2 Detaljnije automatsko testiranje

Iako su osnovni automatski testovi za proveru najvažnijih funkcionalnosti okruženja već napisani, oni se ne pokreću automatski i ciljaju veoma uzan skup funkcionalnosti. Ovaj skup testova bi trebalo proširiti i uvesti princip kontinualne integracije i kontinualnog isporučivanje u proces.

7.2.3 Proširenje funkcionalnosti

Mnoge funkcionalnosti koje su deo prikazanih okruženja, nisu implementirane u okruženju *Pure*. Jedna od većih funkcionalnosti koja nedostaje je rutiranje. Takođe, dobar deo ugrađenih HTML elemenata nemaju svoju implementaciju u okviru okruženja *Pure*. Autor se nada da će barem deo ovih nedostataka biti rešen u budućnosti kroz zajednicu otvorenog koda.

7.2.4 Proširenje šablona Flux na šablon CQRS

Šablon Flux implementiran u okruženju *Pure* predstavlja jednu pojednostavljenu verziju bez okidanja akcija i funkcije *skupljača*. Umesto jedne centralizovane funkcije skupljača, okruženje *Pure* ostavlja odgovornost nenarušavanja konzistentnosti skladišta programerima.

Ovaj mehanizam može da se proširi do šablona CQRS[1] [22] koji bi povećao kompleksnost aplikacija sa jedne strane, ali bi omogućio lakše skaliranje aplikacije u smislu kompleksnosti programskog koda.

Šablon CQRS ima za cilj da u potpunosti odvoji kod koji utiče na promenu stanja skladišta i koda koji samo čita trenutno stanje. Specifičan je u odnosu na šablon *Flux* po tome što padrazumeva i čuvanje redosleda i detalja izvršenih komandi koje menjaju stanja što omogućava repliciranje svakog stanja aplikacije do kog može da se stigne od početnog. Ovaj mehanizam je veoma značajan u pronalaženju eventualnih grešaka u kodu.

7.2.5 Alat za prevođenje HTML-a

Glavnu prepreku za korišćenje novog okruženja u odnosu na postojeće je korišćenje *TypeScript* sintakse umesto HTML-a. Iako ovo ima svojih prednosti koje su već opisane u odeljku 4.2.1, postoji veliki broj postojećih komponenti napisanih u HTML-u, koje je nema razloga pisati ponovo. Alat koji bi prevodio HTML sintaksu u funkcionalne komponente olakšao bi korišćenje okruženja *Pure* programerima koji žele da koriste postojeće komponente napisane u HTML-u.

[1] CQRS - Command Query Responsibility Segregation [22]

Glava 8

Zaključak

U ovom radu predstavljen je razvoj jednog modernog okruženja za razvoj klijentskih veb aplikacija. Pored toga, data je kratka analiza problema postojećih rešenja i najčešćih arhitekturalnih obrazaca.

U okviru rada nalaze se obrazloženja za odabir konkretnih alata i biblioteka, primeri korišćenja okruženja za pravljenje aplikacija kao i objašnjenje implementacije najvažnijih elemenata okruženja.

Ne postoji idealan softver i ne postoji idealno razvojno okruženje. Svaki izbor alata se uvek svodi na optimizaciju jednog skupa kvalitativnih atributa po cenu nekog drugog. Okruženje *Pure* u ovom smislu nije izuzetak. Zbog velikog broja biblioteka i okruženja u ekosistemu *JavaScript* jezika, autor ne očekuje da će se rešenje koristiti u komercijalne svrhe, ali se nada da će ovo rešenje, ili barem neki njegovi delovi, poslužiti kao inspiracija nekim budućim bibliotekama i okruženjima. Okruženje razvijeno u okviru ovog rada objavljeno je pod licencom MIT [14], što znači da je otvoreno za modifikaciju i korišćenje u akademske ili komercijalne svrhe.

Glava 9

Dodatak

Resursi na mreži koji se prilažu uz tekst rada:

- Repozitorijum okruženja *Pure*: `https://github.com/maleksandar/pure-framework`
- Stranica *npm*-paketa „pure-framework”: `https://www.npmjs.com/package/pure-framework`
- Repozitorijum aplikacije „Zdravo svete!”: `https://github.com/maleksandar/pure-framework-hello-world-app`
- Repozitorijum aplikacije „Menadžer zadataka”: `https://github.com/maleksandar/pure-framework-todo-app`
- Online verzija aplikacije „Menadžer zadataka”: `https://pure-framework-todo-demo.netlify.app/`

Bibliografija

[1] Ibtisam Rauf Abdul Majeed. MVC Architecture: A Detailed Insight to the Modern Web Applications Development. *Peer Rev J Sol Photoen Sys*, 1, 2018.

[2] Dan Abramov and Andrew Clark. Redux - A Predictable State Container for JS Apps, 2015. url: `https://redux.js.org/`, visited on: 28/08/2021, repository: `https://github.com/reduxjs/redux`.

[3] ReactiveX contributors. ReactiveX - The Observer pattern done right, 2021. on-line at: `http://reactivex.io/`, visited on: 04/09/2021.

[4] ReactiveX contributors. RxJs - Reactive Extensions Library for JavaScript, 2021. on-line at: `https://rxjs.dev/`, visited on: 04/09/2021.

[5] Roy Thomas Fielding. REST architectural style, 2000. url: `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`, visited on: 28/08/2021.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[7] Veronica Gavrilă, Lidia Bajenaru, and Ciprian Dobre. Modern single page application architecture: A case study. *Studies in Informatics and Control*, 28:231–238, 07 2019.

[8] Facebook Inc. React - A JavaScript library for building user interfaces, 2013. url: `https://reactjs.org/`, visited on: 28/08/2021, repository: `https://github.com/facebook/react/`.

[9] Google Inc. Angular - The modern web developer's platform, 2016. url: `https://angular.io/`, visited on: 28/08/2021, repository: `https://github.com/angular/angular`.

[10] Microsoft Inc. TypeScript - JavaScript with syntax for types, 2012. url: `https://www.typescriptlang.org/`, visited on: 28/08/2021, repository: `https://github.com/microsoft/TypeScript`.

[11] Microsoft Inc. Visual Studio Code - Open source code editor, 2015. url: `https://code.visualstudio.com/`, visited on: 28/08/2021, repository: `https://github.com/microsoft/vscode/`.

[12] Graham Kendall. Would your mobile phone be powerful enough to get you to the moon?, 2019. url: `https://theconversation.com/would-your-mobile-phone-be-powerful-enough-to-get-you-to-the-moon-115933`, visited on: 05/09/2021.

[13] Michael Mathews. JSDoc - API documentation generator for JavaScript, 2021. ulr: `https://jsdoc.app/index.html`, visited on: 07/09/2021.

[14] Aleksandar Milosavljevic. MIT License, 2021. url: `https://github.com/maleksandar/pure-framework/blob/master/LICENSE.md`, visited on: 07/09/2021.

[15] MDN Mozilla Developer Network. Minification, 2020. ulr: `https://developer.mozilla.org/en-US/docs/Glossary/minification`, visited on: 04/09/2021.

[16] MDN Mozilla Developer Network. ECMAScript, 2021. ulr: `https://developer.mozilla.org/en-US/docs/Glossary/ECMAScript`, visited on: 04/09/2021.

[17] MDN Mozilla Developer Network. Prototype Inheritance, 2021. ulr: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`, visited on: 04/09/2021.

[18] MDN Mozilla Developer Network. Truthy, 2021. ulr: `https://developer.mozilla.org/en-US/docs/Glossary/Truthy`, visited on: 04/09/2021.

[19] Robert Netzer and Barton Miller. What are race conditions? - some issues and formalizations. *ACM letters on programming languages and systems*, 1, 09 1992.

[20] NPM. npx - Run a command from a local or remote npm package, 2021. url: `https://docs.npmjs.com/cli/v7/commands/npx`, visited on: 04/09/2021.

[21] Lawrence C. Paulson and Andrew W. Smith. Logic programming, functional programming, and inductive definitions. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*, pages 283–309, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[22] Chris Richardson. Command Query Responsibility Segregation (CQRS), 2021. ulr: `https://microservices.io/patterns/data/cqrs.html`, visited on: 07/09/2021.

[23] OpenJS Foundation Ryan Dahl. Node.js - JavaScript runtime built on Chrome's V8 JavaScript engine., 2009. url: `https://nodejs.org/en/`, visited on: 28/08/2021, repository: `https://github.com/nodejs/node`.

[24] Isaac Z. Schlueter. npm - Package manager for Node.js, 2009. url: `https://www.npmjs.com/`, visited on: 28/08/2021, repository: `https://github.com/npm/npm`.

[25] Open Source. NgRx - Reactive State for Angular, 2021. on-line at: `https://ngrx.io/`, visited on: 04/09/2021.

[26] StackOverflow. 2020 Developer Survey, 2020. url: `https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-professional-developers` visited on: 05/09/2021.