UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Aleksandar Milosavljević

RAZVOJ "PURE" OKRUŽENJA 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: 15. januar 2016.

Svima osim LaTex-u

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

Rezime: Rezime placeholder

Ključne reči: veb, internet, interfejs, html, css, javaskript, razvojno okruženje

Sadržaj

1 Uvod **1**

2 Arhitektura SPA **2**
2.1 Šablon Model-Pogled-Upravljač 3
2.2 Šablon Flux 4

3 Problemi sa popularnim okruženjima **5**
3.1 Problemi sa okruženjem Angular 5
3.2 Problemi sa bibliotekom React 6

4 Okruženje Pure **8**
4.1 Preduslovi 8
4.2 Zdravo svete! 9
4.3 Struktura Pure aplikacija 12
4.4 Aplikacija „Menadžer Zadataka” 15

5 Zahtevi **18**

6 Implementacija okruženja Pure **19**
6.1 Dizajn okruženja Pure 19
6.2 Korišćeni alati, tehnologije i biblioteke 20
6.3 Povezivanje 22
6.4 Funkcionalni Elementi 24
6.5 Komponente 28
6.6 Obrada događaja 31
6.7 Promena stanja 32
6.8 Memoizacija 33

7 Diskusija **34**

8 Zaključak **35**

Bibliografija **36**

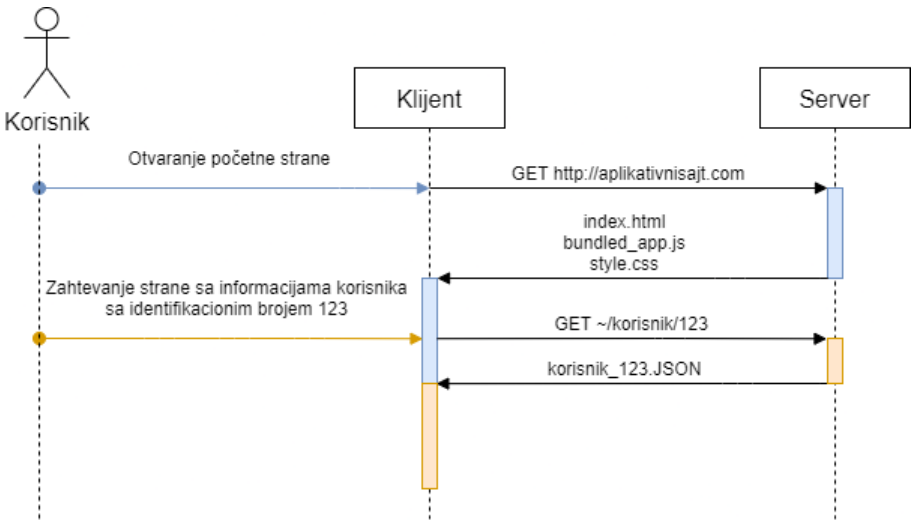
Glava 1

Uvod

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. SPA (eng. *Single Page Application*) arhitektura je dovela ovaj princip do krajnjih granica prebacivanjem celokupne logike prikaza na klijentsku stranu [3]. U ovoj arhitekturi klijent pri prvom zahtevu ka serveru preuzima celokupnu logiku prikaza aplikacije i obrađuje je u pretraživaču, dok svi naredni pozivi služe prostoj razmeni podataka. Ovo je značajno uticalo na samu metodologiju razvoja softvera.



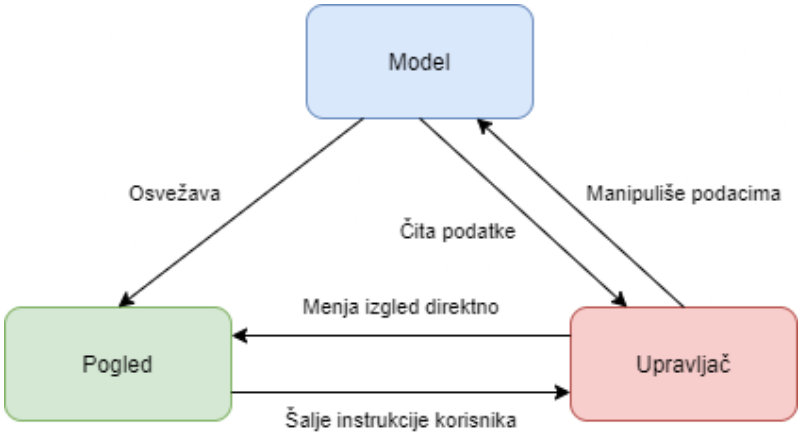
Slika 2.1: Dijagram sekvence SPA arhitekture.

Ovaj pristup omogućio je nezavistan razvoj timovima zaduženim za serversku logiku od timova zaduženih za prezentacionu logiku i vizuelni dizajn. Jedina tačka

spajanja ova dva tima postao je dizajn interfejsa između klijentske i serverske aplikacije. Rešenje ovog problema inženjeri su našli u REST (eng. *Representational State Transfer*) arhitekturalnom šablonu [2]. Odvajanjem razvoja klijentskih i serverskih aplikacija, postalo je moguće razviti složene korisničke aplikacije. 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 perioda višestraničnih aplikacija. Primer jednog takvog šablona je *Model-Pogled-Upravljač* (eng. *Model-View-Controller*)[1].

2.1 Šablon Model-Pogled-Upravljač

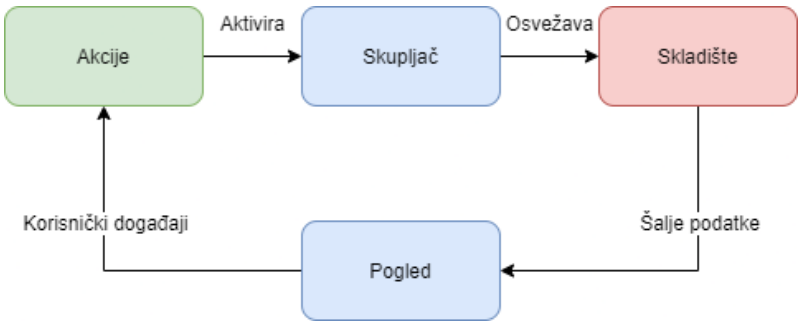
Šablon *Model-Pogled-Upravljač* nastao je pre pojave SPA arhitekture. Koristio se za velike sajtove i aplikacije, ali na takav način da su sve komponente pisane kao serverski kod. Ideja šablona *Model-Pogled-Upravljač* je da omogući razdvajanje odgovornosti između logike podataka, aplikativne logike i vizuelne reprezentacije programa. Ovaj šablon ima istu 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* [5].



Slika 2.2: Arhitektura šablona Model-Pogled-Upravljač

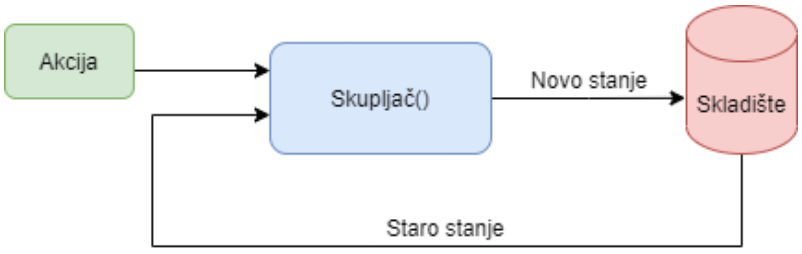
2.2 Šablon Flux

Šablon *Flux* je nastao u kompaniji *Facebook*, iz potrebe inžinjera da otklone probleme sa deljenjem stanja između različitih komponenti i čestim stanjem utrkivanja (eng. *race condition* [8]).



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 stanja utrkivanja. *Pogled*, kao i kod šablona *Model-Pogled-Upravljač*, služi za opis vizualne reprezentacije programa. *Pogled* utiče na tok podataka tako što pri korisničkom događaju odabira *akciju* koju će da emituje ka *skupljaču* (eng. *reducer*). *Skupljač* je čista funkcija [9], koja kao argumente prihvata akciju i trenutno stanje skladišta i vraća novo stanje skladišta.



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 koji sloj *kada* može upravljati i pristupati podacima. Biblioteka *React* nastala je u kompaniji *Facebook* kao predstavnik ovog šablona [4].

Glava 3

Problemi sa popularnim okruženjima

Iako su moderna razvojna okruženja poput okruženja *Angular* i biblioteke *React* omogućila programerima da napišu veoma kompleksne korisničke aplikacije, to ne znači da ova okruženja nemaju svoje mane i 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 inžinjera i testera. Šablon *Model-Pogled-Upravljač* koji prestavlja osnovni arhitekturalni dizajn okruženja Angular, značajno olakšava rad velikom broju ljudi na istoj kodnoj bazi.

Upravljanje stanjem

Problem sa okruženjem *Angular* nastaje kada među različitim komponentama delimo stanje. Stanje utrke vremenom počinje da se javlja sve češće kao problem, a „pametna krpljenja" počinju da zauzimaju sve veći deo kodne baze.

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

Logika u pogledu

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

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

Prikaz koda 3.1: Isečak Angular koda

Kod prikazan u isečku 3.1 predstavlja primer za izlistavanje stavki (`item`) iz niza `items`. Drugim rečima, ovo nije HTML[1] element, već jedna instrukcija napisana *Angular* jezikom za opisivanje pogleda, koja će u vremenu izvršavanja biti zamenjena sa pravim HTML elementom. 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], jer logika ispisana ovde prolazi kroz više faza prevođenja i uglavnom je izvršava optimizovani i minifikovani kod samog okruženja.

3.2 Problemi sa bibliotekom React

Okruženje *React* razvijeno je od strane *Facebook* inžinjera 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 stanja utrkivanja, ali su neki drugi problemi ipak ostali deo dizajna.

Razdvajanje odgovornosti

U prikazima 3.2 i 3.3 vidimo primer preuzet sa zvanične veb stranice *React* biblioteke. U pitanju su dva različita načina definisanja *React* komponente. U primeru 3.2 je prikazan način koji koristi *JSX* sintaksu. *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* [6]). 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 povezanosti vizualne

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

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 transpajlira ovaj kod na čist *JavaScript* (prikaz 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.

Iako je ovo veoma zgodan mehanizam, 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 element, 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 ovakvih, 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*), u smislu da ga je gotovo nemoguće debagovati.

```
1 const element = (
2   <h1 className="greeting">
3     Hello, world!
4   </h1>
5 );
```

Prikaz koda 3.2: Fajl `ToDoApp.ts`

```
1 const element = React.
    createElement(
2   'h1',
3   {className: 'greeting'},
4   'Hello, world!'
5 );
```

Prikaz koda 3.3: Fajl `ToDoApp.ts`

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.

Glava 4

Okruženje Pure

Kako bismo se upoznali detaljnije sa novim okruženjem, krenućemo od jednostavnog primera, „Zdravo svete" (eng. *„Hello world"*) aplikacije.

4.1 Preduslovi

Kako bismo mogli da pokrenemo *Pure* aplikaciju neophodno je da radno okruženje *Node.js*[10] bude instalirano na računaru. 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*[6]. Autor preporučuje alat otvorenog koda, razvijen od strane Microsoft inžinjera: *Visual Studio Code*[7].

Okruženje *Pure* može se preuzeti preko *npm* [2] sistema [11]. To možemo uraditi na dva načina:

1. Instaliranjem paketa `pure-framework` u već postojeći *npm* modul. (konzolna komanda: 4.1)
2. Pokretanjem skripte za pravljenje `hello-world` projekta, bez prethodnog instaliranja *npm* paketa. (konzolna komanda: 4.2)

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

4.2 Zdravo svete!

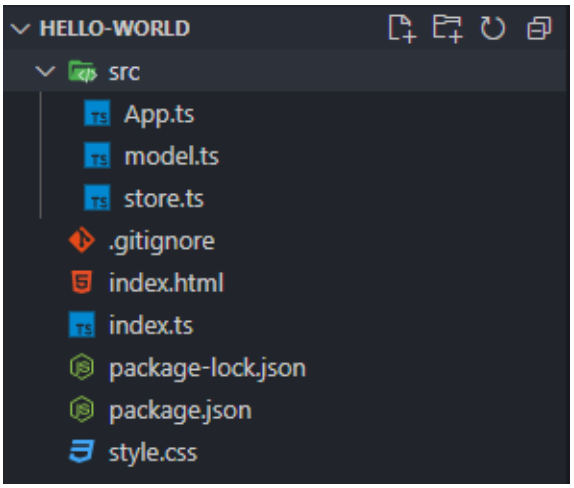
Ukoliko već imamo postojeći *npm* modul na kojem radimo, možemo u okviru njega instalirati `pure-framework` komdanom:

```
$ npm install pure-framework
```
(4.1)

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

```
$ npx pure-framework hello-world
```
(4.2)

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 4.1.



Slika 4.1: Grafikon

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

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

```
$ cd hello-world && npm install
```
(4.3)

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

```
$ npm run start
```
(4.4)

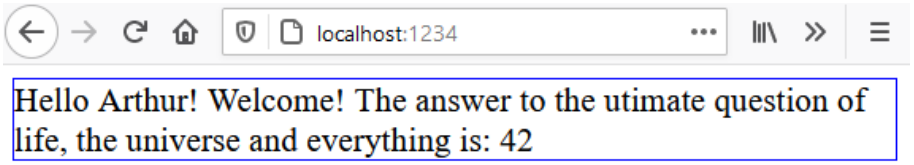
Ukoliko pokretanje ove komande nije izbacilo grešku, konzola bi trebalo da nas obavesti da je server za razvoj pokrenut i da možemo da mu pristupimo na simboličkoj adresi `http://localhost:1234` (konzolni odgovor 4.1)

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

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

Konzolni odgovor 4.1: Konzolna poruka nakon pokretanja aplikacije

Ukoliko u pretraživaču otvorimo stranicu `http://localhost:1234`, trebalo bi da vidimo stranicu koja izgleda kao na slici 4.2



Slika 4.2: „Zdravo svete” aplikacija, pokrenuta na lokalnom serveru.

Struktura aplikacije

Fajl koji sadrži centralnu logiku naše aplikacije je `src/App.ts` (Prikaz 4.2)

```
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);
```

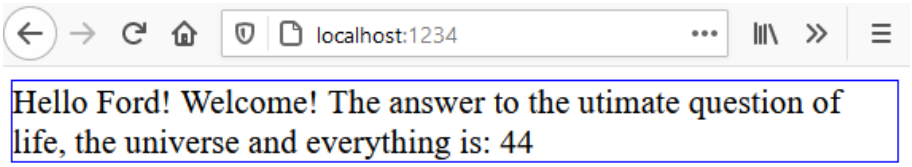
Prikaz koda 4.2: Sadržaj fajla `App.ts`

Ukoliko pogledamo aplikaciju u pretraživaču (slika 4.2) videćemo da struktura HTML elemenata odgovara strukturi koju vraća funkcija `template()`.

Recimo da želimo da dodelimo ponašanje našoj aplikaciji, tako da svaki put kada kliknemo na element povećamo broj iz pozdravne za jedan, a ime promenimo na „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 }
```

Prikaz koda 4.3: Fajl `App.ts` nakon dodate funkcionalnosti



Slika 4.3: Hello World aplikacija, nakon dodavanja funkcionalnosti.

Ukoliko otvorimo ponovo našu aplikaciju u pretraživaču 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 4.3).

4.3 Struktura Pure aplikacija

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

Mesto povezivanja

Prvi fajl koji se učitava i izvršava, nakon fajla `index.html` je `index.ts` (Prikaz 4.4). 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 sa *Pure* aplikacijom.
2. Konstruktorsku funkciju korene komponente napisane u okruženju *Pure*.
3. Referencu na objekat koji upravlja skladištem (eng. *storage / store*).

Treći argument služi tome da okruženje može da osluškuje promene stanja u objektu `store` i tako zna kada treba da obnovi iscrtavanje elemenata.

```
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);
```

Prikaz koda 4.4: Fajl `index.ts`

Komponente

Komponente sadrže logiku i vizuelni opis segmenta 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 direktno, 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 sadrži vizuelnu strukturu komponente i po potpisu funkcije mora da vrati objekat tipa `FunctionalElement`. Tip `FunctionalElement` je interfejs koji implementiraju sve funkcije iz kolekcije ugrađenih funkcionalnih elemenata (`div()`, `span()`, `h1()`, ...). Važno je napomenuti da i klasa `Component<T>` implementira interfejs `FunctionalElement`, što znači da povratna vrednost metode `template()` takođe može biti i druga komponenta.

Funkcionalni elementi

Sve ugrađene funkcionalne elemente, koji služe opisu ugrađenih HTML elemenata, mogu se uvesti sa putanje `,,pure-framework/html''`. Ugrađeni funkcionalni elementi (funkcije uvežene iz `,,pure-framework/html''`) poštuju 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 4.5 i 4.6 možemo videti kako se prevodi jedna struktura funkcionalnih elemenata u HTML elemente.

```
div({ class: 'app-root'}, [
  span('First span'),
  span('Second span'),
  div({ class: 'inner-div'}, [
    span([
      span('This span is left
          from'),
      italic(' italic text'),
      span(', and this one is on
          the right of it.'),
    ])
  ])
]);
```

Prikaz koda 4.5: Funkcionalni element

```
<div class="app-root">
  <span>First span</span>
  <span>Second span</span>
  <div class="inner-div">
    <span>
      <span>This span is left
          from</span>
      <i> italic text</i>
      <span>, and this one is on
          the right of it.</span
          >
    </span>
  </div>
</div>
```

Prikaz koda 4.6: Rezultujući HTML

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 „Zdravo svete" aplikacije može se videti na prikazu 4.7.

```
export interface AppModel {
  name: string;
  answer: number;
}
```

Prikaz koda 4.7: Fajl `src/model.ts`

Skladište koje se koristi u okviru okruženja Pure zasnovano je na mehanizmu *BehaviorSubject*-a[4] 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

[4] *BehaviorSubject* - eng. Subjekt ponašanja

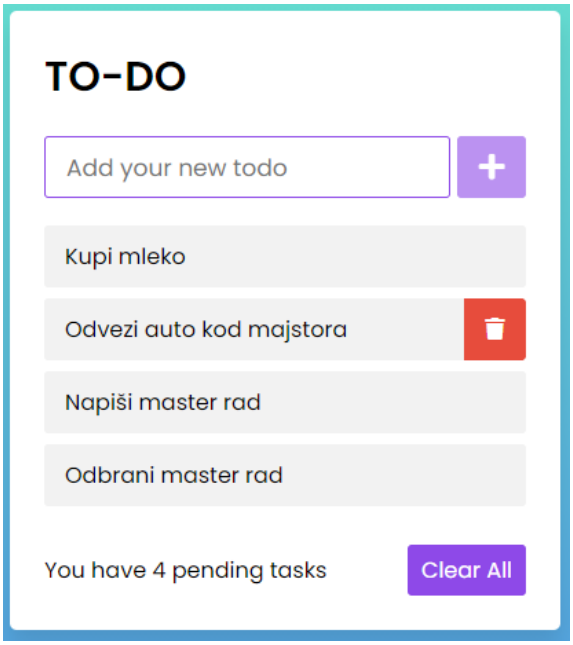
objekat podrazumevanog stanja koji odgovara po svojoj strukturi prosleđenom generičkom argumentu (Prikaz 4.8).

```
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 })
```

Prikaz koda 4.8: Fajl `src/model.ts`

4.4 Aplikacija „Menadžer Zadataka”

Pogledaćemo sada malo kompleksniji primer. U pitanju je aplikacija za upravljanje dnevnim obavezama[5]. Online verzija ove aplikacije nalazi se na stranici `https://pure-framework-todo-demo.netlify.app/` (Slika 4.4). Izvorni kod aplikacije nalazi se na adresi:
`https://github.com/maleksandar/pure-framework-todo-app`



Slika 4.4: Online verzija TO-DO aplikacije

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

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);
```

Prikaz koda 4.9: Fajl `ToDoApp.ts`

Ukoliko pogledamo strukturu liste potomaka u metodi `template()` (Prikaz 4.9) videćemo da ona upravo opisuje tri vizualna segmenta aplikacije 4.4:

1. Zaglavlje sa naslovom, poljem za unos novog zadatka i dugmetom za potvrdu unosa (Funkcija `headerSegment()`).
2. Segment sa listom aktuelnih zadatka (Funkcija `todoSegment()`).
3. Podnožje sa prikazom zbirnih informacija i dugmetom za brisanje liste aktuelnih zadataka (Funkcija `footerSegment()`).

Na isečku koda 4.9 prikazan je kod funkcije koja opisuje segment sa listom aktuelnih zadataka. Funkcija `todoSegment()` vraća funckcionalni element `ul()` (Koji odgovara `<ul>` HTML elementu). Na liniji 48 vidimo poziv *spread (eng. raširi)* operatora nad nizom koji se dobija kao rezultat mapiranja niza `this.state.todoList` u niz `li` elemenata. Svaki od `li` elemenata 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 na liniji 54, prikaza 4.9. Primetimo bitan detalj na liniji 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 `todoApp` komponente i `store` objekta ostvarena je pozivanjem metode `bootstrap()` u fajlu `index.ts` (Prikaz 4.4).

Pogledajmo sada šta sadrži objekat `store`, koji uvozimo iz fajla `src/store.ts` (Prikaz 4.8).

Na osnovu prikaza 4.8 vidimo da je `store` objekat 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` (Prikaz 4.10)

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

Prikaz koda 4.10: Fajl `src/model.ts`

Glava 5

Zahtevi

Pre upuštanja u detalje implementacije okruženja *Pure* razmotrimo zahteve koje bi jedno novo razvojno okruženje trebalo da ispuni.

1. Kako bi izbeglo probleme iz poznatijih okruženja kao što su **logika u pogledu** (Odeljak 3.1) i **razdvajanje odgovornosti** (Odeljak 3.2) okruženje *Pure* ne sme da uvodi novi, niti koristi bilo kakav postojeći specijalizovani jezik za opisivanje HTML-a. Dodatni zahtev u pogledu opisa HTML strukture komponenenata je mogućnost debagovanja logike i strukture izgleda komponente.
2. Kako bi se izbegao problem sa deljenjem stanja (Odeljak 3.1), *Pure* treba da se oslanja na šablon *Flux* i jednosmeran tok podataka. Stanje mora biti centralizovano, a interfejs za upravljanje stanjem mora biti robustan i konzistentan među komponentama.
3. Kako bi što manje grešaka došlo do korisnika, potrebno je uočiti što više grešaka koje je moguće detektovati u fazi razvoja.
4. Distribucija razvojnog okruženja *Pure* mora biti dostupna kroz javni repozitorijum *npm* paketa.
5. Okruženje *Pure* mora biti otvoreno za modifikaciju.
6. Komponente okruženja *Pure* moraju imati jednostavnu i lako čitljivu strukturu.

Glava 6

Implementacija okruženja Pure

U ovom poglavlju pogledaćemo detalje implementacije okruženja *Pure* i pojasniti neke odluke u dizajnu okruženja.

Ime okruženja *Pure*[1] inspirisano je funkcionalnom paradigmom i čistim funkcijama. 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 mehanizam memoizacije (eng. *Memoization*), 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*.

6.1 Dizajn okruženja Pure

Okruženje Pure preuzima šablon *Flux* od biblioteke *React*, ali inspiraciju za konretnu implementaciju ovog šablona preuzima od biblioteke *NgRx* koja je deo *Angular*-ekosistema. Od okruženja *Angular* preuzima pristup razvoju koji podrazumeva pisanje komponenti u jeziku *TypeScript* umesto jezika *JavaScript*.

Zašto ne HTML?

Iako je HTML oproban alat za opisivanje strukture dokumenta, on ima svoja ograničenja koja nije tako lako prevazići. Programi za obradu teksta pri pisanju

[1] *Pure* - eng. Čisto, čist

HTML fajla uglavnom teško mogu da zaključe kontekst povezivanja pojedinačnih HTML-elemenata sa konkretnim funkcijama i promenljivima napisanim u *JavaScript*-u i zbog toga je podrška alata, a samim tim i iskustvo tokom pisanja HTML koda, značajno lošija 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 jeziku ne postoji opcija za uključivanje podkomponenti. Korišćenjem *TypeScript* funkcija umesto HTML sintakse rešavaju se ovi problemi. Alati za upravljanje *TypeScript* kodom značajno su napredniji od alata za upravljanje HTML-om.

6.2 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 budu do sada nerešeni (ili nedovoljno adekvatno rešeni) problemi, potrebno je odabrati i uvezati ove postojeće alate na pravi način. U ovom poglavlju ćemo opisati glavne alate i biblioteke koje predstavljaju deo okruženja *Pure*, ili su korišćene tokom razvoja okruženja.

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 funkcija očekuje nisku kao tip podataka, a mi nad tim argumentom pozivamo metodu koja nije definisana na prototipu niske (REFERENCA: Prototipsko nasleđivanje), kompajler će nam izbaciti grešku i upozoriti nas da to (verovatno) nije metoda koju smo želeli da pozovemo.

Pošto *TypeScript* predstavlja suštinski samo nadskup jezika JavaScript (korišćenje tipova je opciono), možemo jednostavno balansirati između striktnih tipova i sigurnosti sa jedne strane i čitljivosti koda sa druge, tako da je cena uvođenja ove tehnologije u okruženje veoma niska, a dobici mogu biti značajni.

Parcel

Parcel je popularan *bundler*[2] za veb aplikacije. *Bundler* u razvojnim okruzenjima ima dva zaduženja:

1. Pravljenje *produkcionog JavaScript* fajla. sa minifikovanim i „poruznjenim" kodom (REFERENCA: minification & uglificatrion).
2. Povezivanje koda i veb pretraživača tokom pisanja koda.

Produkcioni JavaScript fajl je fajl koji sadrži celokupnu logiku aplikacije, ali je za potrebe brzog preuzimanja i učitavanja od strane pretraživača preveden na nižu verziju EcmaScript standarda (zbog starijih verzija pretraživača), minifikovan i „poružnjen". Ukratko, sve beline iz koda su uklonjene, a varijable preimenovane u kraća simbolička imena.

Glavna prednost *bundler*-a *Parcel* je u minimalnoj neophodnoj konfiguraciji. Konkretan razlog zbog kog je *Parcel* odabran kao *bundler*, je taj što *Parcel* automatski prevodi *TypeScript* kod na *JavaScript* bez ikakve dodatne konfiguracije, pa zbog toga možemo u okviru `index.html` fajla direktno da zahtevamo *TypeScript* fajl (primer 6.1).

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

Prikaz koda 6.1: Uključivanje *TypeScript* fajla direktno u `index.html`

RxJs

RxJs je biblioteka koja sadrži generičku implementaciju šablona *Observer*[3] za *JavaScript*. Deo je većeg skupa alata otvorenog koda pod nazivom *ReactiveX* koji služe reaktivnom programiranju. 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. Okruženje *Pure* koristi biblioteku *RxJs* iz istih razloga, ali u značajno pojednostavljenoj varijanti.

[2] *Bundler* - eng. Upakivač, Zapakivač. Onaj koji pakuje.
[3] *Observer* - eng. Posmatrač

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 pretraživača 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 *JavaScript* objekata.

6.3 Povezivanje

Kao što je već pomenuto u sekciji 4.3, `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 `src/bootstrap.ts` i može se videti u primeru 6.2.

```
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 }
```

Prikaz koda 6.2: Fajl `ToDoApp.ts`

Funkcija `bootstrap()` je generička funkcija koja prihvata jedan argument tipa, koji odgovara tipu korenog modela podataka, i tri standardna argumenta. Standardni argumenti su:

1. Referenca na element DOM-a.
2. Konstruktorska funkcija komponente.
3. Referenca na objekat skladišta.

Ovi argumenti su već opisani u sekciji 4.3, pa njihovo značenje nećemo ovde ponovo navoditi.

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

1. Instanciranje korene komponente pozivom konstruktorske funkcije (Linja 10).
2. Osluškivanje promene stanja skladišta (Linija 11).
3. Registracija anonimne funkcije za ponovno iscrtavanje elemenata DOM-a (Linije 11-16).

Iako će pojedinačni detalji komponenti, 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.

Primetimo da konstruktorsku funkciju, koju `bootsrap()` funkcija prihvata kao drugi argument, 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 6.7.

6.4 Funkcionalni Elementi

Tip `FunctionalElement` predstavlja interfejs najopštijeg nivoa u hijerarhiji strukturnih elemenata okruženja *Pure*[4], 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 }
```

Prikaz 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.
4. Referencu na „roditelja" DOM-elementa (`null` u slučaju korene komponente).

Metodu `render()` koja se poziva pri iscrtavanju DOM-elemenata. Važno je napomenuti da *TypeScript* interfejsi mogu definisati i polja, a ne samo metode[5].

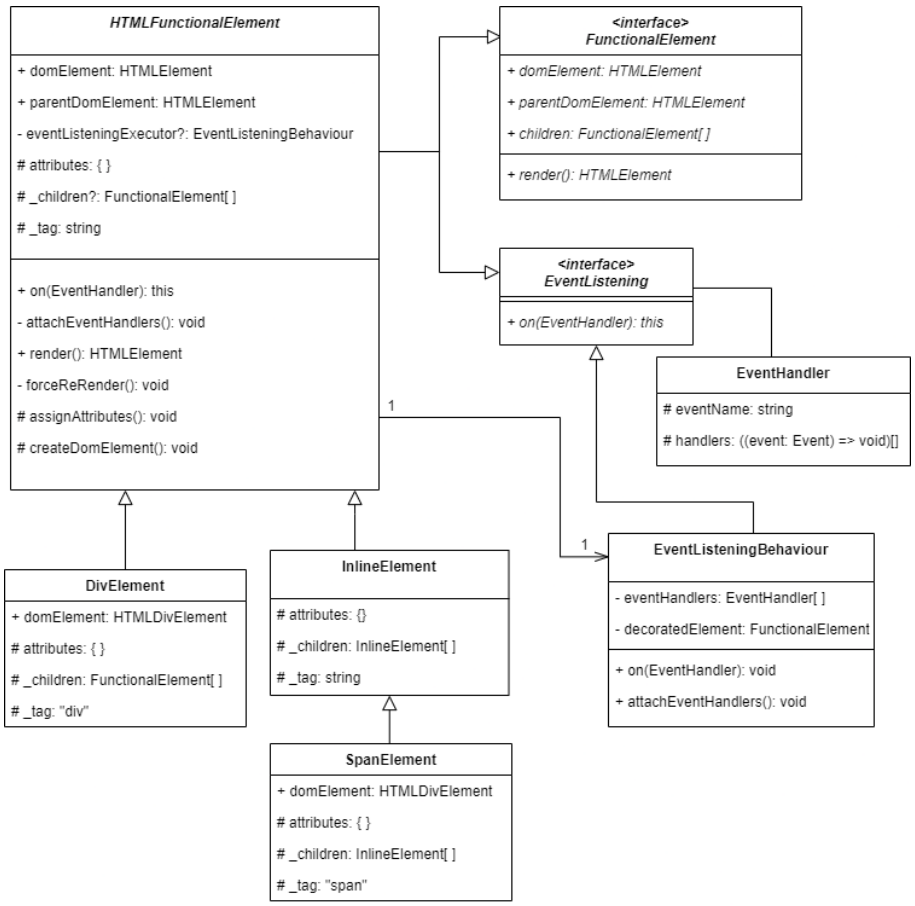
Ugrađeni Funkcionalni Elementi

Kako bismo izgradili nove komponente koristeći šablon Kompozicije (eng. *Composition Pattern*), moramo se osloniti na neke postojeće, to jest, element 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

[4]Elementi za opis DOM-strukture

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

aplikacije [reftorepo],



Slika 6.1: Dijagram klasa za klasu `HTMLFunctionalElement`

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

```
1 import { FunctionalElement, HTMLFunctionalElement } from '/core';
2 export class DivElement extends HTMLFunctionalElement {
3     constructor(protected attributes: {}, protected _children:
          FunctionalElement[]) {
4         super(attributes, _children, 'div')
5     }
6 }
```

Prikaz 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`.

```
1  import { EventListening } from "./eventListening";
2  import { EventListeningBehaviour } from "./eventListeningBehaviour";
3  import { FunctionalElement } from "./functionalElement";
4
5  export abstract class HTMLFunctionalElement implements
       FunctionalElement, EventListening {
6    public domElement: HTMLElement;
7    public parentDomElement: HTMLElement;
8    private _eventListeningExecutor: EventListeningBehaviour = null;
9
10   constructor(protected attributes: {}, protected _children:
       FunctionalElement[], protected _tag: string) {
11     if (arguments.length === 2) {
12       // in case attributes object is provided as a first argument
13       this.attributes = arguments[0];
14       this._children = arguments[1];
15     }
16     else if (arguments.length === 1) {
17       // in case we are provided with only one argument - we asume it
             is the array of children or a single child
18       this._children = arguments[0];
19     }
20     this._eventListeningExecutor = new EventListeningBehaviour(this);
21   }
22
23   public on(event: string, ...handlers: ((event: Event) => void)[]) {
24     this._eventListeningExecutor.on(event, ...handlers);
25     return this;
26   }
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      if (this.attributes) {
46        Object.keys(this.attributes).forEach(attribute => {
47          if (this.attributes[attribute]) {
48            this.domElement.setAttribute(attribute, this.attributes[
                attribute]);
49          }
50        });
51      }
52    }
53
54    protected createDomElement(): void {
55      this.domElement = document.createElement(this._tag);
56      if (!this.children || this.children.length === 0) {
57        return;
58      }
59      this.children.forEach(child => {
60        child.parentDomElement = this.domElement;
61        this.domElement.appendChild(child.render());
62      });
63    }
64
65    private attachEventHandlers() {
66      this._eventListeningExecutor.attachEventHandlers();
67    }
68  }
```

Prikaz 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()`).

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*[6] i dodeljuje prosleđene atribute DOM-elementu.

Mehanizam registrovanja osluškivača događaja i samo upravljanje događajima biće obrađeno u sekciji (REFERENCA).

6.5 Komponente

Komponente u okruženju *Pure* predstavljaju funkcionalne elemente koji se dinamički mogu 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.

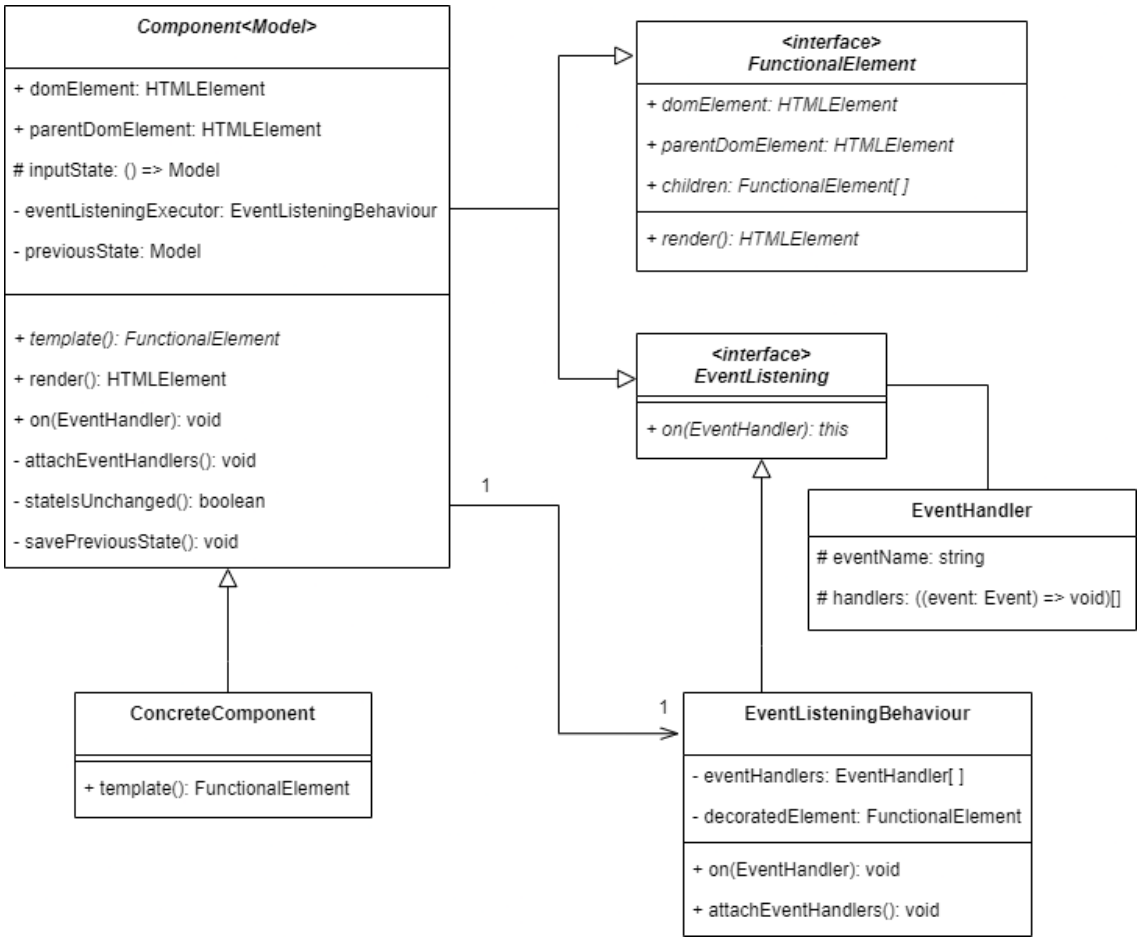
Metoda `template()` mora da vrati tip koji implementira interfejs `FunctionalElement`. To može biti druga komponenenata ili neki od ugrađenih funkcionalnih elemenata.

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()`, a rezultat tog poziva se vraća kao izlazna vrednost metode `render()`.

Pogledajmo sada implementaciju klase `Component<T>`.

```
1 import { FunctionalElement } from "./functionalElement";
2 import { EventListeningBehaviour } from "./eventListeningBehaviour";
3 import { areEqual, cloneDeep } from "../utils";
4 import { EventListening } from "./eventListening";
5
```

[6]Skup vrednosti u *JavaScript*-u koje se konverzijom prevode na *boolean* vrednost *TRUE*



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

```
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  };
```

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

6.6 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 okruženja *Pure* `FunctionalElement` obe klase implementiraju interfejs `EventListening` i sadrže instancu klase koja direktno 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) šablona *Dekorator* razdvajamo implementaciju za osluškivanje i reagovanje na korisničke događaje.

Pogledajmo detalje ovog interfejsa i konkretne implemntacije (ista implementacija interfejska se koristi i za `HTMLFunctionalElement` i za `Component<Model>`).

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

Prikaz 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  }
```

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

6.7 Promena stanja

Pogledajmo sada kako možemo dinamički izmeniti izgled naše aplikacije pomoću promene stanja skladišta.

```
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 }
```

Prikaz koda 6.9: Fajl `core/eventListeningBehaviour.ts`

Skladište u razvojnom okruženju *Pure* se zasniva na `BehaviorSubject<T>` tipu iz biblioteke *RxJs*. `BehaviorSubject<T>` je objedinjuje posmatrača i subjekta

posmatranja u šablonu *Observer*[7]. Drugim rečima, nad `Subject` tipom (natklasa tipa `BehaviorSubject` definsina u biblioteci *RxJs*) je objekat nad kojim možete da registruje funkciju za obradu događaja pri promeni stanja (metoda `.subscribe(...)`), ali i objkat nad kojim možete pozvati metodu za promenu stanja (metoda `.next(...)`).

Ovaj mehanizam omogućava nam da imamo centralizovan sistem za skladištenje podataka. Pogledajmo kod konkretne implementacije.

Videćemo da klasa ima tri javne metode:

1. Metodu za dohvatanje objekta nad kojim vršimo subskripciju (`state$`)
2. Metodu za vraćanje trenutnog stanja.
3. Metodu za promenu stanja.

6.8 Memoizacija

[7] *Observer* - eng. Posmatrač

Glava 7

Diskusija

Glava 8

Zaključak

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] 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.

[3] 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.

[4] 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/`.

[5] 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`.

[6] 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`.

[7] 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/`.

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

[9] 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.

[10] 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`.

[11] 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`.