

Kreiranje prve aplikacije korišćenjem Vanilla JS-a

Ukoliko se prvi put susrećete sa pojmom iz naslova (*Vanilla JavaScript*), možda ćete pomisliti da će u nastavku biti ilustrovano korišćenje nekog posebnog softverskog okvira, odnosno biblioteke, koja se zove *Vanilla JS*. Ipak, stvarnost je nešto drugačija, pa ukoliko se do sada niste susretali sa ovim pojmom, sadržina lekcije pred vama može vas iznenaditi.

Šta je Vanilla JS?

Iako po svom nazivu podseća na ime JS biblioteke, *Vanilla JS* je pojam koji se ne odnosi ni na kakvu JavaScript biblioteku. Reč je zapravo o pojmu koji se koristi da označi čist JavaScript kod, koji se kreira bez upotrebe bilo kakvih biblioteka ili softverskih okvira.

Pojam *Vanilla JS* izmislio je 2012. godine Erik Vastl (Eric Wastl), kao šalu koja treba da podseti frontend programere da je i bez upotrebe popularnih i modernih biblioteka i softverskih okvira moguće kreirati JavaScript aplikacije. Erik Vastl napravio je i kompletan web sajt sa šaljivim sadržajem, koji *Vanilla JS* opisuje u maniru modernih JavaScript biblioteka:

<http://vanilla-js.com/>

Na prikazanoj adresi možete da pročitate da je *Vanilla JS* brz, lagan, multiplatformski softverski okvir koji omogućava razvoj neverovatno moćnih JavaScript aplikacija. Na sajtu *Vanilla JS*a postoji i sekcija koja omogućava preuzimanje takve *neverovatne* biblioteke, a kada obavite preuzimanje `vanilla.js` fajla, moći ćete da vidite da je, sasvim očekivano, njegova sadržina prazna.

Šalu na stranu, ideja Erika Vastla je zaista kreativna i podseća na jednu veoma važnu činjenicu – korišćenjem čistog JavaScript koda moguće je kreirati sve ono što i korišćenjem modernih JavaScript softverskih okvira. Ono što je još značajnije, odlično poznavanje JavaScripta i modernih aplikativnih programskih interfejsa web pregledačaapsolutno su preduslovi za korišćenje i razumevanje biblioteka i softverskih okvira koji su danas popularni.

Danas se veoma često, pogotovo u oglasima za radne pozicije, mogu sresti pojmovi Angular programer, React programer ili nešto slično. Takve pojmove uglavnom formulisu osobe koje ne pripadaju programerskom miljeu, sve sa ciljem da akcenat stave na neku od danas popularnih JavaScript biblioteka. Na taj način, početnici u svetu frontend programiranja mogu se navesti na pogrešan put i pomisliti da je dovoljno naučiti korišćenje neke popularne JavaScript biblioteke kako bi se kvalifikovali za bavljenje nekim od veoma dobro plaćenih poslova. Stvarnost je ipak znatno drugačija. Uspešan Angular programer na prvom mestu mora biti dobar Frontend JavaScript programer ili jednostavno programer.

Na kraju JavaScript biblioteke i softverski okviri se smenjuju, dok su JavaScript i skup Web API-ja konstante koje se neprekidno usavršavaju i razvijaju. Na primer, nekada je jQuery bio apsolutni standard frontend razvoja. Vremena se menjaju, pa konstantnim

usavršavanjem JavaScript jezika u poslednje vreme jQuery gubi na popularnosti. Tome, na primer, pogoduje i činjenica da je GitHub u potpunosti izbacio jQuery sa frontenda web sajta GitHub.com (slika 2.1).

The screenshot shows a tweet from Mislav Marohnić (@mislav) with a profile picture. The tweet text is: "We're finally finished removing jQuery from [GitHub.com](#) frontend. What did we replace it with? No framework whatsoever:" followed by a bulleted list: • querySelectorAll, • fetch for ajax, • delegated-events for event handling, • polyfills for standard DOM stuff, • CustomElements on the rise. Below the tweet is a large GitHub logo. The tweet has 8,977 likes and was posted at 11:57 AM - Jul 25, 2018. It also indicates 3,626 people are talking about this.

Slika 2.1. Objava na Twitteru u kojoj se govori o uklanjanju jQuery biblioteke sa web sajta GitHub.com

Na kraju, možemo rezimirati da je preduslov za korišćenje različitih JavaScript biblioteka, kako onih koje su popularne danas tako i onih koje će se koristiti u budućnosti, odlično poznavanje JavaScript jezika i Web API-ja.

Da li su nam JavaScript biblioteke uopšte potrebne?

Prethodni redovi vas mogu navesti na zaključak da su JavaScript biblioteke i softverski okviri suvišni u današnjem modernom frontend razvoju. To naravno nije tačno. JavaScript biblioteke i softverski okviri obezbeđuju sledeće veoma važne aspekte razvoja:

- olakšavaju pisanje kompleksnog koda,
- ubrzavaju pisanje koda i razvoj aplikacija,
- omogućavaju da se fokusiramo na prave vrednosti aplikacije, odnosno na ono što našu aplikaciju razlikuje od neke druge, umesto trošenja vremena na implementaciju,
- olakšavaju timski rad zato što nameću korišćenje standardizovanih pristupa i dobrih praksi.

Zbog svega ovoga u nastavku ove lekcije biće prikazano kreiranje jedne jednostavne JavaScript aplikacije korišćenjem Vanilla JavaScripta, pri čemu će biti demonstrirani osnovni postulati JavaScript jezika i Web API-ja. U narednim modulima, kreiranje JavaScript aplikacija biće ilustrovano i upotrebom popularnih JavaScript biblioteka i softverskih okvira.

Pitanje

Vanilla JS je:

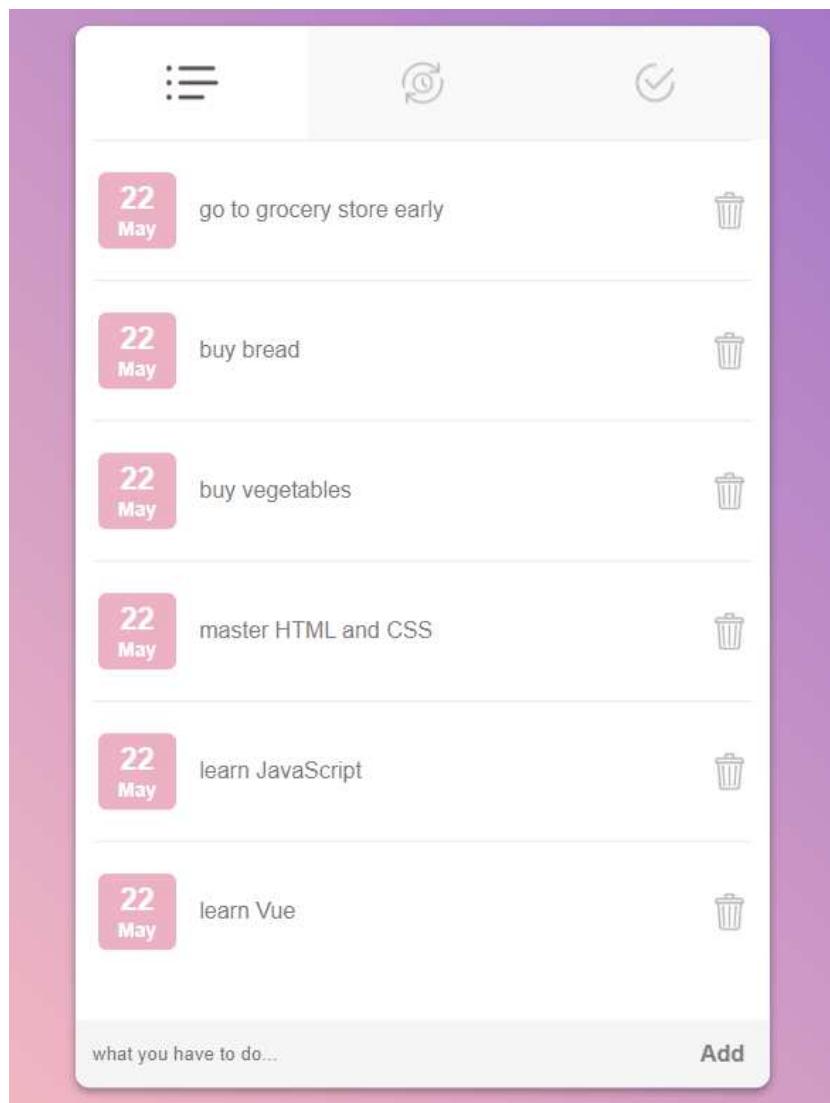
- popularna JavaScript biblioteka
- biblioteka za kreiranje serverske logike web aplikacija
- pristup koji podrazumeva pisanje čistog JavaScript koda, bez upotrebe pomoćnih biblioteka**
- arhitekturalni softverski šablon

Objašnjenje:

Vanilla JS je pojam koji se koristi da označi čist JavaScript kod, bez upotrebe bilo kakvih biblioteka ili softverskih okvira.

ToDo aplikacija

U lekciji pred vama biće kreirana jedna jednostavna aplikacija koja korisnicima treba da omogući kreiranje podsetnika. Stoga će aplikacija imati naziv *ToDo*, a izgledaće kao na slici 2.2.



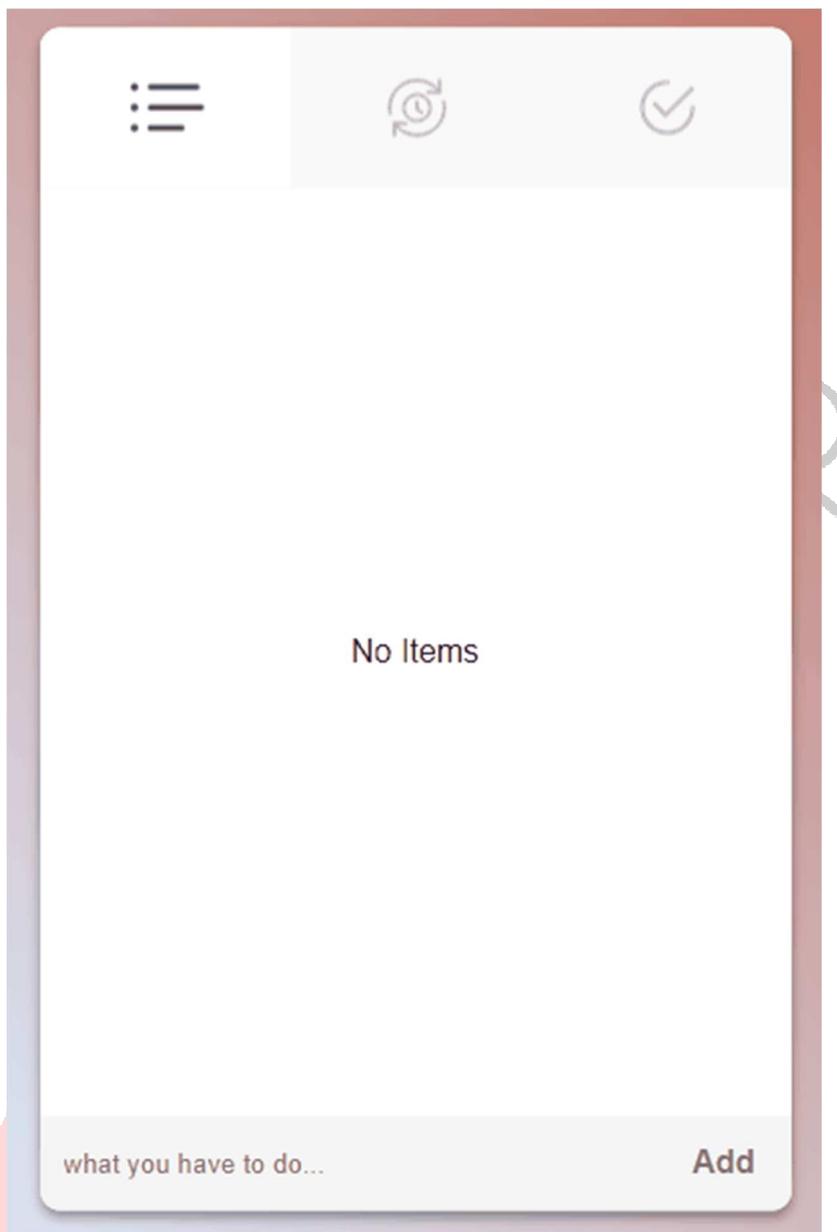
Slika 2.2. Izgled ToDo aplikacije

Korisničko okruženje aplikacije će konceptualno da bude sastavljeno iz sledećih oblasti:

- zaglavlje (engl. *header*) – sekcija sa tri dugmeta za odabir prikaza (sve stavke, kompletirane stavke, stavke koje još nisu kompletirane),
- glavna sekcija, odnosno telo aplikacije – sekcija unutar koje će se prikazivati odgovarajuće stavke, u zavisnosti od toga šta korisnik odabere u zaglavlu,
- podnožje (engl. *footer*) – sekcija sa poljem za unos nove stavke i dugmetom za pokretanje upisivanja.

Klikom na neku od stavki korisnik će imati mogućnost da ciklično menja njen status iz aktivne u kompletiranu i obratno. Takođe, korisnici će imati mogućnost i da u potpunosti obrišu svaku pojedinačnu stavku, klikom na dugme za brisanje.

Aplikacija će funkcionisati kao što je to prikazano animacijom 2.1.



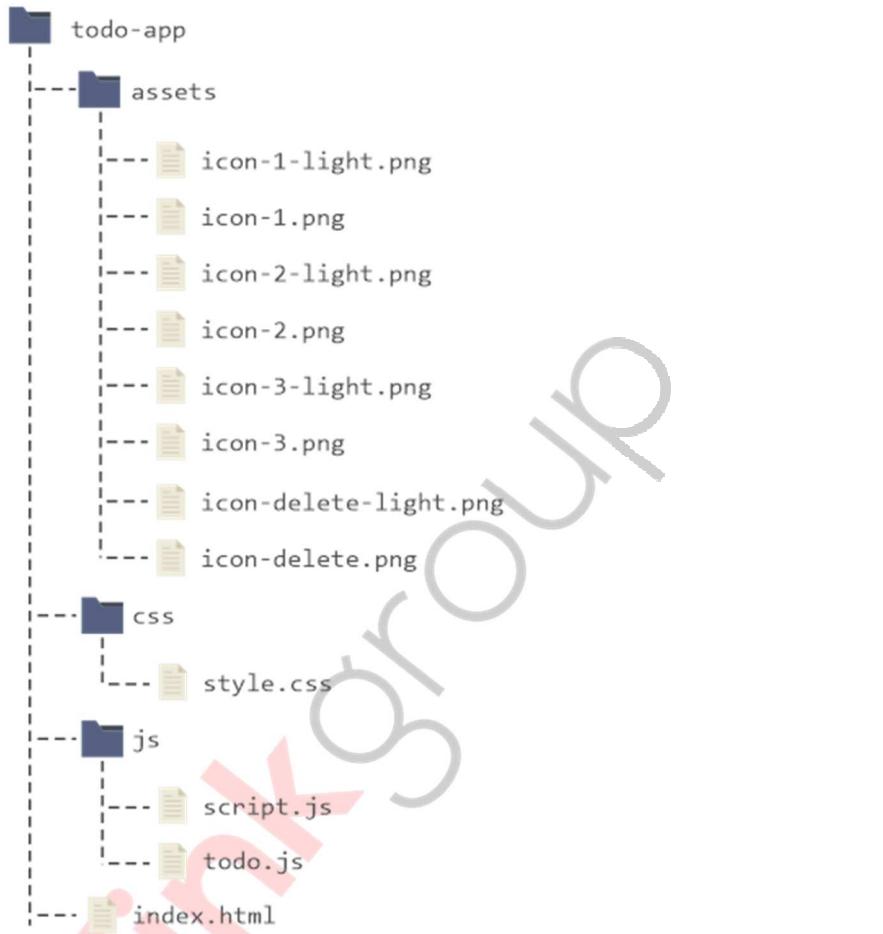
Animacija 2.1. Osnovni način funkcionisanja ToDo aplikacije

ToDo aplikacija će stavke čuvati unutar Local Storage skladišta, tako da će unete stavke biti dostupne i nakon zatvaranja pregledača, odnosno sve dok se keš memorija web pregledača eksplicitno ne obriše.

Kod za kreiranje ToDo aplikacije možete da preuzmete sa sledećeg linka:

[todo-app.rar](#)

Kada raspakujete preuzetu arhivu, moći ćete da vidite da fajl strukturu ToDo aplikacije, koja izgleda kao na slici 2.3.



Slika 2.3. Fajl struktura ToDo aplikacije

Namena različitih fajlova i foldera je sledeća:

- assets – folder za smeštanje resursa; reč je o PNG slikama koje se unutar aplikacije koriste kao ikonice; ToDo poseduje ukupno 4 ikonice; svaka ikonica postoji u dve varijante – svetloj i tamnoj,
- css – folder za smeštanje stilizacije; kompletna stilizacija se nalazi unutar jednog fajla – style.css,
- js – folder za smeštanje JavaScript koda; JavaScript kod je podeljen u dva fajla:
 - todo.js – JavaScript fajl sa osnovnom logikom ToDo aplikacije,
 - script.js – JavaScript fajl sa logikom za obradu globalnih događaja i za integraciju ToDo funkcionalnosti,
- index.html – HTML dokument ToDo aplikacije.

Ukoliko pogledate fajlove iz `js` foldera, moći ćeće da vidite da su oni prazni. S obzirom na to da je tema ovoga kursa razvoj JavaScript aplikacija, kreiranje programske logike ToDo aplikacije biće prikazano u nastavku ove lekcije. Stoga iskoristite HTML i CSS osnovu iz preuzetih fajlova, a dodavanje preostalog koda obavlajte praćenjem nastavka ove lekcije.

Mobile-first i Single-page Application

ToDo aplikacija koja se kreira u ovoj lekciji koristiće neka od načela Single-page modela razvoja, tako da ćeće u nastavku na realnom primeru videti na koji način je SPA moguće kreirati korišćenjem Vanilla JS-a.

Takođe, kreiranje ToDo aplikacije na način koji ilustruje ova lekcija predstavlja klasičan primer Mobile-first razvoja, s obzirom na to da je korisničko okruženje aplikacije razvijeno za mobilne uređaje. Takva osnova se nakon toga može prilagoditi i za ostale uređaje (laptop, desktop...).

Struktura i stilizacija

S obzirom na to da HTML i CSS nisu primarna tema ovoga kursa, u nastavku se nećemo posebno baviti kodom za markiranje i stilizovanje, kako bi više prostora preostalo za priču o JavaScript logici koja pokreće našu ToDo aplikaciju. Kao što je rečeno u prethodnim redovima, iskoristite HTML i CSS iz arhive koja se može preuzeti sa nešto ranije navedenog linka. Ipak, kako biste lakše razumeli JavaScript koji bude pisan u nastavku, ukratko ćemo se upoznati sa osnovnom strukturom prezentacije i kodom za stilizovanje. HTML struktura naše ToDo aplikacije izgleda ovako:

```
<div id="app-container">

    <header>
        <div class="header-cell" id="tab-1">
        </div>

        <div class="header-cell" id="tab-2">
        </div>

        <div class="header-cell" id="tab-3">
        </div>
    </header>

    <div id="app-body">

        <div id="all-items" class="items-container">
            <div id="all-items-container">
                <span class="no-items-message">No Items</span>
            </div>
        </div>

        <div id="active-items" class="items-container">
            <div id="active-items-container">
                <span class="no-items-message">No Items</span>
            </div>
        </div>
    </div>
</div>
```

```
<div id="completed-items" class="items-container">
  <div id="completed-items-container">
    <span class="no-items-message">No Items</span>
  </div>
</div>

</div>

<footer>
  <input id="content" type="text" placeholder="what you have to
do..."> <span id="add-btn">Add</span>
</footer>

</div>
```

Osnovna struktura prezentacije podeljena je na tri celine:

- `<header></header>` – zaglavlj je sa tri dugmeta za odabir liste stavki.

Unutar zaglavlja se nalaze tri dugmeta koja su kreirana korišćenjem `div` elemenata. Ikonice su definisane kao pozadinske slike. Bitno je primetiti da svaki od elemenata koji simuliraju dugme poseduje i `id` atribut sa jedinstvenom vrednošću. Takve `id` vrednosti će u nastavku da budu korišćene za identifikaciju dugmića od JavaScript logike.

- `<div id="app-body"></div>` – telo za prikaz izabrane liste stavki

Telo ToDo aplikacije sačinjeno je iz tri zasebna `div` elementa. Svaki `div` element predstavlja po jedan tab koji će biti aktiviran klikom na odgovarajuće dugme iz zaglavlja. Sadržaj tela će u nastavku da bude dinamički postavljan unutar unutrašnjih `div` elemenata, unutar kojih se sada nalazi po jedan `span` element sa tekstrom *No Items*.

- `<footer></footer>` – podnožje sa poljem za unos nove stavke

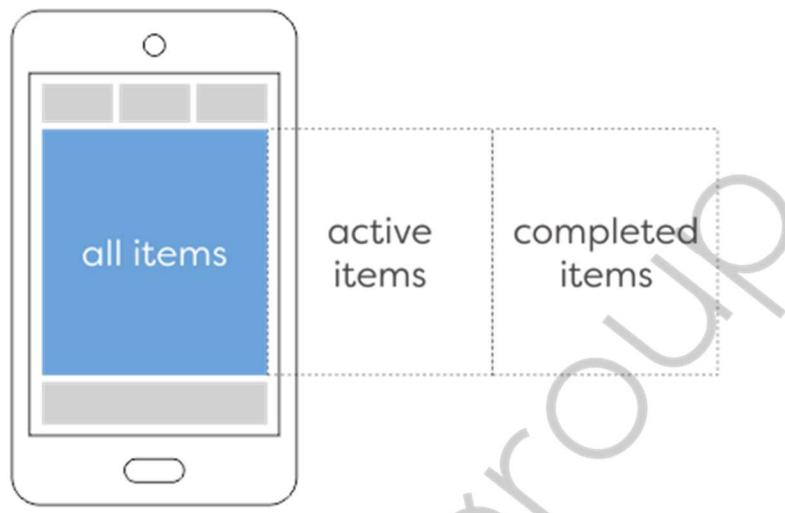
Unutar podnožja ToDo aplikacije nalazi se `input` element za unos sadržaja nove stavke i dugme za pokretanje unosa.

Realizacija funkcionalnosti tabova

Pre nego što započnemo da pišemo programsku logiku ToDo aplikacije, potrebno je realizovati funkcionalnost koja će omogućiti kretanje kroz različite prikaze. ToDo aplikacija poseduje ukupno tri različita prikaza:

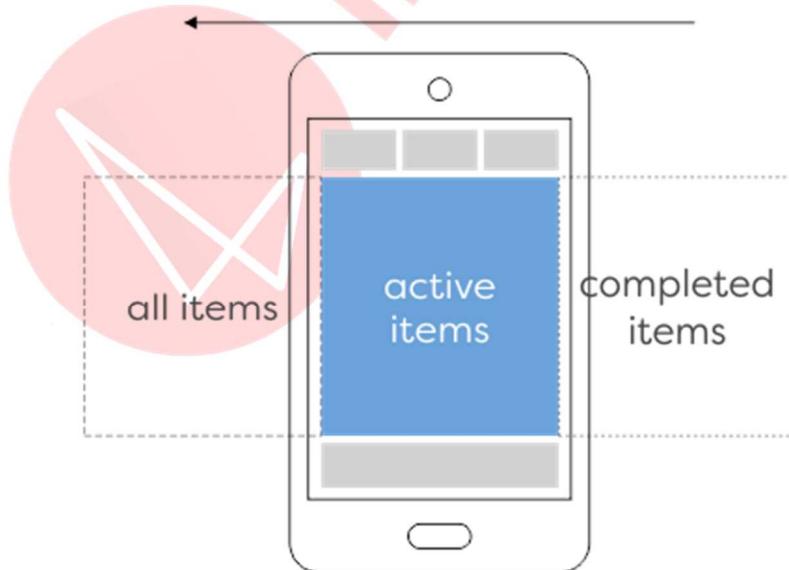
- sve stavke,
- aktivne stavke,
- kompletirane stavke.

Liste za prikaz odgovarajućih stavki realizovane su kao zasebni `div` elementi, koji su korišćenjem Flexboxa raspoređeni u jednom redu, ali tako da svaka lista uvek zauzima kompletну širinu aplikacije. Na taj način se osigurava da će u jednom trenutku uvek moći da bude prikazana samo jedna lista (slika 2.4).



Slika 2.4. Struktura prikaza lista stavki

Na slici 2.4. se može videti kako su složeni susedni `div` elementi za prikaz različitih ToDo stavki. Stoga će klikom na odgovarajuće dugme unutar zaglavlja zapravo da budu pomerana ovakva tri `div` elementa uлево. Na primer, za prikaz aktivnih stavki, svi elementi će da budu pomereni uлево за једну ширину aplikације (slika 2.5).



Slika 2.5. Logika za promenu tabova

Sve ovo je u praksi realizovano sledećim JavaScript kodom:

```
var headerButtons = document.getElementsByClassName('header-cell');
var allItems = document.getElementById("all-items");

for (let i = 0; i < headerButtons.length; i++) {
    headerButtons[i].onclick = function() {
        openTab(parseInt(this.id.substr(this.id.length - 1)));
    }
}
```

Prikazanim JavaScript kodom prvo se dolazi do div elemenata koji predstavljaju dugmiće unutar zaglavlja (elementi sa klasom .header-cell) i do elementa kojim se predstavljaju sve ToDo stavke (#all-items).

Za svaki .header-cell element definiše se funkcija za obradu click događaja. Tako će klikom na neku od stavki unutar zaglavlja da bude aktivirana funkcija openTab(), kojoj se prosleđuje jedna numerička, celobrojna vrednost. Funkcija openTab() može da prihvati brojeve 1, 2 ili 3, koji ukazuju na tab koji je potrebno otvoriti. Kako bi se znalo koji je tab potrebno otvoriti, čita se id vrednost elementa (dugmeta) na koji je korisnik kliknuo. Elementi imaju id atribute tab-1, tab-2 i tab-3, a prikazanim kodom se korišćenjem funkcije substr() dolazi do numeričke vrednosti (1, 2 ili 3).

Funkcija za otvaranje odgovarajućeg taba izgleda ovako:

```
function openTab(no) {

    document.querySelectorAll('.header-cell').forEach(item => {
        item.classList.add("inactive-header-cell");
    });

    document.getElementById("tab-" + no).classList.remove("inactive-
header-cell");

    switch (no) {
        case 1:
            allItems.style.marginLeft = "0%";
            break;
        case 2:
            allItems.style.marginLeft = "-100%";
            break;
        case 3:
            allItems.style.marginLeft = "-200%";
            break;
    }
}
```

Funkcija za otvaranje tabova je logički podeljena na dva dela. Prvo se postavlja odgovarajuća vizuelna reprezentacija dugmića unutar zaglavlja. To se obavlja tako što se prvo na svako dugme unutar zaglavlja postavlja .inactive-header-cell klasa, koja sadrži stilizaciju dugmeta u neaktivnom stanju. Zatim se klasa .inactive-header-cell uklanja sa dugmeta na koje je korisnik kliknuo. Time se postiže definisanje odgovarajućeg izgleda dugmića unutar zaglavlja.

Nakon definisanja odgovarajućeg izgleda dugmića unutar zaglavlja, prelazi se na prikaz odabrane liste ToDo stavki. U zavisnosti od prosleđene vrednosti, div elementi za prikaz stavki pomeraju se uлево definisanjem negativne leve margine na prvom elementu (elementu za prikaz svih stavki). Za prikaz prve liste, margina se postavlja na 0. Središnja lista se prikazuje tako što se na prvu listu postavlja leva margina od -100%, pa se svi elementi pomeraju za jednu kompletну širinu uлево. Na kraju, za prikaz poslednje liste (liste kompletiranih stavki), leva margina se postavlja na -200%, čime se div elementi pomeraju za dve širine uлево.

Definisanjem upravo prikazane logike unutar naše ToDo aplikacije biće moguće kretati se kroz tabove kojima se predstavljaju sve stavke, aktivne stavke i kompletirane stavke, respektivno.

Rutiranje

Veoma važan aspekt razvoja JavaScript aplikacija jeste rutiranje. Rutiranje je zapravo prisutno uvek, odnosno prilikom korišćenja svakog web sajta, ali se ono kod tradicionalnih, višestraničnih web sajtova obavlja bez naše intervencije, odnosno obavlja ga web pregledač.

U svom najjednostavnijem obliku rutiranje je uskladišvanje odgovarajućeg resursa sa URL adresom. Ipak, u zavisnosti od tipa web sajta, rutiranje može postojati u oblicima koji se znatno razlikuju.

Najjednostavniji primer rutiranja spomenut je u prethodnim redovima. Reč je o rutiranju prilikom korišćenja web sajtova sa statičkim HTML dokumentima. Svaki od dokumenata poseduje svoju URL adresu, pri čemu je svaka takva URL adresa osnovni način na koji se dolazi do konkretnе stranice. Na primer, klikom na link koji vodi na putanju www.mysite.com/about, unutar web pregledača prikazuje se sadržaj about.html stranice. Tako je ovo klasičan primer rutiranja koje automatski obavlja web pregledač.

Kod web aplikacija sa stranicama koje se dinamički generišu na serveru, a pogotovo kod Single-page aplikacija, proces rutiranja je znatno kompleksniji, a veoma često se ne može realizovati isključivo korišćenjem web pregledača.

Kod web aplikacija sa stranicama koje se dinamički generišu na serveru, uloga servera je da zahtev za nekim resursom u formi URL adrese obradi i isporuči odgovarajući odgovor. Web pregledač dobija ono što je zatražio bez ikakvog znanja o tome kako je isporučeni sadržaj nastao. U takvim slučajevima, iz ugla klijenta i dalje sve funkcioniše kao i kod tradicionalnih, statičkih web sajtova, zato što apstrakciju rutiranja skriva web server.

Kod JavaScript aplikacija, a pre svih kod onih koje oponašaju native aplikacije korišćenjem Single-page modela, rutiranje gotovo u potpunosti potпада pod oblast interesovanja frontend programera. Da je to stvarno tako, možete se uveriti i na primeru naše ToDo aplikacije, koja se razvija u ovoj lekciji. U prethodnim redovima prikazano je kako se realizuje prelazak sa jednog na drugi tab. Unutar ToDo aplikacije to je jedini segment koji korisniku omogućava navigaciju, odnosno prelazak sa pogleda na pogled. Ipak, mogli ste da vidite da takva navigacija ne funkcioniše kao navigacija korišćenjem klasičnih link elemenata. Drugim rečima, promena aktivnog taba ni na koji način nije propraćena promenom URL adrese, što znači da trenutno naša ToDo aplikacija ne poseduje rutiranje. Nešto ranije je rečeno da je kod Single-page aplikacija rutiranje u nadležnosti frontend programera, pa će stoga naredni redovi biti posvećeni postizanju veoma jednostavnog rutiranja na primeru naše ToDo aplikacije.

Rutiranjem mi želimo da povežemo tri taba ToDo aplikacije sa jedinstvenim URL adresama, tako da se zadovolji sledeće:

- klikom na neko dugme unutar zaglavlja biće otvoren odgovarajući tab, a unutar adresne trake web pregledača biće upisana odgovarajuća URL adresa, baš kao da je korisnik kliknuo na neki link,
- unosom URL adrese koja predstavlja neki od tabova ToDo aplikacije automatski će biti otvoren odgovarajući tab.

Kako bi se postiglo sve ono što je opisano, neophodno je da delimično izmenimo logiku koju smo već kreirali, ali i da dodamo novu. Prvo će biti prikazana logika koja će, na osnovu odabране stavke u zaglavlju, da ažurira URL adresu:

```
function openTab(no) {  
  
    document.querySelectorAll('.header-cell').forEach(item => {  
        item.classList.add("inactive-header-cell");  
    });  
  
    document.getElementById("tab-" + no).classList.remove("inactive-  
header-cell");  
  
    switch (no) {  
        case 1:  
            allItems.style.marginLeft = "0%";  
            window.location.hash = '#all-items';  
            break;  
        case 2:  
            allItems.style.marginLeft = "-100%";  
            window.location.hash = '#pending-items';  
            break;  
        case 3:  
            allItems.style.marginLeft = "-200%";  
            window.location.hash = '#active-items';  
            break;  
    }  
}
```

Unutar `openTab()` funkcije sada su dodate tri nove naredbe za definisanje URL hasha. To se postiže korišćenjem jednog posebnog Web API-ja koji se zove Location API.

Location API

Location API jeste naziv za skup funkcionalnosti web pregledača koji omogućava rukovanje URL adresama na klijentu. Location je objekat kojim se predstavlja URL adresa trenutno učitane web stranice. Takvom objektu je moguće pristupiti korišćenjem svojstava `document.location` i `window.location`. To praktično znači da je svojstvu `location` moguće pristupiti i bez navođenja prefiksa, s obzirom na to da je sastavni deo `Document` objekta.

Location objekat poseduje brojna svojstva koja se mogu koristiti za dolazak do izdvojenih delova od kojih je sačinjen URL:

- `href` – kompletna URL adresa u string obliku; definisanjem vrednosti ovoga svojstva preusmerava se pregledač na definisani adresu,
- `protocol` – protokol iz URL adrese,
- `host` – naziv hosta i broj porta,
- `hostname` – naziv domena,
- `port` – broj porta,
- `pathname` – putanja do resursa koja uključuje i uvodni / karakter,
- `search` – query string sa uvodnim karakterom upitnika (?),
- `hash` – identifikator fragmenta sa uvodnim hash (#) karakterom,
- `origin` – objedinjene informacije o nazivu hosta, portu i protokolu.

Sva upravo navedena svojstva, osim svojstva `origin`, omogućavaju i čitanje i pisanje. Svojstvo `origin` omogućava samo čitanje, s obzirom na to da unutar sebe objedinjuje tri različita segmenta URL adrese.

Location objekat poseduje i nekoliko metoda:

- `assign()` – preusmerava web pregledač na URL adresu prosleđenu kao parametar,
- `reload()` – osvežava trenutnu stranicu; ukoliko se prosledi vrednost `true`, stranica se uvek dobavlja sa servera; ukoliko se `true` vrednost ne navede, ne garantuje se učitavanje sa servera, već pregledač stranicu može da isporuči i iz keš memorije,
- `replace()` – zamenjuje trenutnu stranicu onom koja se nalazi na URL adresi koja je prosleđena kao parametar; na prvi pogled ova metoda ima identičan efekat kao i metoda `assign()`; ipak razlika je u istoriji posećenih stranica; metoda `replace()` upisuje novu stranicu preko postojeće unutar istorije pregleda, te tako u nastavku neće biti moguće vratiti se na prethodnu stranicu odabirom komande `Back`,
- `toString()` – vraća string vrednost URL adrese.

Nakon ove jednostavne intervencije nad `openTab()` metodom, klik na neke od dugmića iz zaglavlja biće proračen dodavanjem odgovarajućeg identifikatora unutar URL adrese.

U sklopu rutiranja, pored već realizovanog ponašanja, neophodno je korisniku omogućiti i da odgovarajući tab otvorи direktnom intervencijom nad URL adresom, odnosno unosom odgovarajućeg hash segmenta. To će biti postignuto korišćenjem sledeće logike:

```
window.addEventListener( "hashchange" , function(e) {  
    examineHash();  
});  
  
examineHash();  
  
function examineHash() {  
    switch (window.location.hash) {  
        case '#all-items':  
        case "":  
            openTab(1);  
            break;  
        case '#pending-items':  
            openTab(2);  
            break;  
        case '#active-items':  
            openTab(3);  
            break;  
    }  
}
```

Za dobijanje dojave o promeni hash segmenta URL adrese, u primeru se sluša događaj hashchange.

Događaj hashchange

hashchange je događaj koji se aktivira kada dođe do promene identifikatora fragmenta unutar URL adrese. Drugim rečima, događaj hashchange se aktivira kada se unutar URL-a promeni deo koji je definisan nakon karaktera hash (#).

Prilikom promene identifikatora fragmenta unutar URL adrese i prilikom prvog učitavanja stranice, aktivira se funkcija examineHash(). Unutar nje se čita hash vrednost i u zavisnosti od pročitane vrednosti otvara se odgovarajući tab, pozivanjem već prikazane openTab() metode.

Bitno je primetiti da je unutar examineHash() funkcije definisano da će se prvi tab otvarati i kada unutar URL adrese ne postoji identifikator fragmenta, tako što je identična logika definisana za prva dva slučaja switch uslovnog bloka.

Kreiranje programske logike ToDo aplikacije

Kompletna programska logika ToDo aplikacije biće smeštena unutar jednog zasebnog fajla (todo.js) i enkapsulirana unutar jedne JavaScript funkcije, čime se na najlakši način kreira jedinstveni prostor imena:

```
var ToDo = function (rootElementAll, rootElementActive,  
rootElementCompleted) {  
  
    //ToDo logic  
  
}
```

Možete videti da konstruktorska funkcija `ToDo()` prihvata tri parametra, koji se koriste za definisanje DOM elemenata unutar kojih je potrebno prikazivati ToDo stavke. Zamišljeno je da korisnik ovakve funkcije prvo kreira ToDo objekat i da tom prilikom dostavi elemente unutar kojih je potrebno da se prikazuju stavke. Ono što korisnik prosledi smešta se unutar tri objektna svojstva:

```
var ToDo = function (rootElementAll, rootElementActive, rootElementCompleted) {  
    this.rootElementAll = rootElementAll;  
    this.rootElementActive = rootElementActive;  
    this.rootElementCompleted = rootElementCompleted;  
}
```

Unutar ToDo objekta postojaće i dve konstruktorske funkcije za kreiranje objekata koji će biti korišćeni tokom rada ToDo aplikacije:

```
var ToDo = function (rootElementAll, rootElementActive, rootElementCompleted)  
{  
  
    this.rootElementAll = rootElementAll;  
    this.rootElementActive = rootElementActive;  
    this.rootElementCompleted = rootElementCompleted;  
  
    let ToDoItem = function (content, date) {  
        this.id = Math.random().toString(36).substring(7);  
        this.content = content;  
        this.date = date;  
        this.completed = false;  
    }  
  
    let ToDoItemViewModel = function (toDoItem, views) {  
        this.data = toDoItem;  
        this.views = views;  
    }  
  
    let ToDoItems = [];  
}
```

`ToDoItem()` je konstruktorska funkcija za kreiranje klasičnih objekata modela. Drugim rečima, korišćenjem `ToDoItem()` funkcije biće kreirani objekti koji predstavljaju stavke ToDo aplikacije. Svaki objekat stavke biće predstavljen sledećim svojstvima:

- `id` – jedinstveni identifikator stavke; automatski se generiše prilikom kreiranja `ToDoItem` objekta,
- `content` – tekst (sadržina) stavke,
- `date` – datum kreiranja stavke,
- `completed` – svojstvo koje čuva informaciju o tome da li je stavka kompletirana ili nije.

Stavke ToDo aplikacije, koje će se kreirati tokom funkcionisanja aplikacije, biće smeštane unutar niza imenovanog kao `todoItems`. Možete da vidite da je vrednost takve promenljive za sada inicijalizovana jednim praznim nizom.

Generisanje novih stavki

Najznačajnija funkcionalnost ToDo aplikacije jeste mogućnost generisanja novih stavki. Stavke će da budu kreirane na osnovu šablona koji će prvo biti definisan u `string` obliku:

```
const TODO_ITEM_TEMPLATE = `<div class="todo-item-date">
    <span class="day">Nov</span>
    <span class="month">2020</span>
</div>
<div class="todo-item-content">
    <span class="data">Lorem ipsum dolor sit amet, consectetur
adipiscing elit.</span>
</div>
<span class="delete-btn" title="delete"></span>
`;
```

Unutar jedne konstante smešten je HTML kod kojim će se predstavljati pojedinačne stavke ToDo aplikacije.

Template literals

Za definisanje HTML koda stavki ToDo aplikacije koristi se relativno nova funkcionalnost JavaScript jezika, koja se naziva *Template literals*. Takva funkcionalnost podrazumeva kreiranje stringova korišćenjem specijalnih karaktera navodnika – ````. Kada se `string` definiše između ovakvih navodnika, moguće je prilikom njegovog formiranja upotrebiti nekoliko naprednih pristupa. Jedan od takvih pristupa jeste i mogućnost definisanja `string` vrednosti u više redova, bez potrebe za bilo kakvom konkatenacijom.

Ovakav šablon koristiće metoda `generateToDoItemView()`:

```
function generateToDoItemView(toDoItem) {
    //first, create root element
    let todoItemRoot = document.createElement('div');
    todoItemRoot.classList.add("todo-item");
    todoItemRoot.setAttribute('data-id', toDoItem.id);
    todoItemRoot.innerHTML = TODO_ITEM_TEMPLATE;

    todoItemRoot.getElementsByClassName("day")[0].innerHTML =
        toDoItem.date.toLocaleString('default', { day: 'numeric' });
    todoItemRoot.getElementsByClassName("month")[0].innerHTML =
        toDoItem.date.toLocaleString('default', { month: 'short' });
    var dataElem =
        todoItemRoot.getElementsByClassName("data")[0].innerHTML = toDoItem.content;
        todoItemRoot.getElementsByClassName("delete-
        btn")[0].setAttribute('data-id', toDoItem.id);

    todoItemRoot.classList.add(toDoItem.completed ? "completed" : null);
```

```

let ToDoItemRootCopy = ToDoItemRoot.cloneNode(true);

rootElementAll.append(ToDoItemRoot);

if (ToDoItem.completed) {
    rootElementCompleted.append(ToDoItemRootCopy);
} else {
    rootElementActive.append(ToDoItemRootCopy);
}

let ToDoItemViewModel = new ToDoItemViewModel(ToDoItem,
[ToDoItemRoot, ToDoItemRootCopy]);
ToDoItems.push(ToDoItemViewModel);

//register handlers for delete button
registerDeleteHandlers(ToDoItemViewModel);

//register handlers for click on item
registerClickHandlers(ToDoItemViewModel);
}

```

Metoda `generateToDoItemView()` kao svoj parametar prihvata jedan `ToDoItem` objekat koji se kreira korišćenjem nešto ranije prikazane konstruktorske funkcije `ToDoItem()`. Reč je o objektu kojim se predstavlja jedna `ToDo` stavka.

Unutar metode `generateToDoItemView()` obavljaju se sledeći poslovi:

1. prvo se kreira koren DOM element kojim će biti predstavljena jedna stavka; na takav element se postavljaju odgovarajuće klase i atributi, a zatim se kao njegov sadržaj postavlja šablon koji je definisan u string obliku i predstavljen nešto ranije,
2. bitno je primetiti definisanje `data-id` atributa na korenom elementu stavke; za njegovu vrednost se postavlja `id` stavke, što će kasnije biti korišćeno kao osnovni mehanizam za povezivanje HTML elemenata stavke sa objektom kojim se takve stavke predstavljaju unutar aplikativne logike,
3. korišćenjem kreirane instance korenog elementa jedne `ToDo` stavke zatim se dolazi do određenih elemenata koji nose sadržaj i, u zavisnosti od osobina prosleđenog objekta koji predstavlja stavku, šablon se popunjava podacima,
4. kreirani DOM objekat se zatim klonira korišćenjem metode `cloneNode()`; na taj način se dobijaju dva potpuno identična DOM objekta, zato što će svaka stavka unutar `ToDo` aplikacije da bude predstavljena korišćenjem dva HTML elementa; jedan element će uvek postojati unutar liste svih stavki, dok će se drugi element nalaziti ili unutar liste aktivnih ili unutar liste kompletiranih stavki, naravno u zavisnosti od toga da li je stavka markirana kao kompletirana ili nije markirana,
5. sledeći korak jeste dodavanje kreiranih DOM objekata u odgovarajuće kontejnerske elemente,
6. kreira se `ToDoItemViewModel` objekat unutar koga se čuvaju podaci o objektu stavke i o dva HTML elementa kojima je on predstavljen unutar aplikacije,
7. `ToDoItemViewModel` objekat se smešta unutar niza `ToDoItems`,
8. pozivaju se metode za registrovanje događaja za markiranje stavki i za njihovo brisanje.

Metoda `generateToDoItemView()` završava se pozivanjem dve metode kojima se registruje logika koja je potrebno da se aktivira kada korisnik inicira markiranje ili brisanje ToDo stavke.

Brisanje stavki

Metoda za registrovanje logike brisanja izgleda ovako:

```
function registerDeleteHandlers(toDoViewModel) {  
    for (let i = 0; i < ToDoViewModel.views.length; i++) {  
        ToDoViewModel.views[i].getElementsByClassName("delete-btn")[0].onclick = function (e) {  
  
            e.stopPropagation();  
  
            var id = this.dataset.id;  
            var index = ToDoItems.findIndex(item => item.data.id === id);  
  
            if (index > -1) {  
                //remove from array and from DOM  
                ToDoItems.splice(index, 1);  
  
                ToDoViewModel.views[0].parentNode.removeChild(ToDoViewModel.views[0]);  
  
                ToDoViewModel.views[1].parentNode.removeChild(ToDoViewModel.views[1]);  
            }  
        }  
    }  
}
```

Unutar metode `registerDeleteHandlers()` prolazi se kroz niz DOM objekata koji upućuju na elemente kojima se predstavljaju stavke i obavlja se registrovanje logike koja će se aktivirati prilikom pojave `click` događaja za `delete-btn` element koji poseduje svaka stavka. Kada korisnik klikne na dugme za brisanje neke stavke, biće obavljen sledeće:

1. zaustavlja se propagiranje događaja, kako se klikom na dugme za brisanje ne bi aktivirale funkcije za obradu događaja definisane na roditeljskim elementima; tako se sprečava aktiviranje funkcije koja obraduje markiranje stavke, kada korisnik pokrene brisanje,
2. dolazi se do vrednosti `data-id` atributa kojim se jednoznačno određuje ToDo stavka,
3. vrednost `data-id` atributa se koristi za pronađak ToDo stavke unutar `ToDoItems` niza elemenata,
4. element kojim se predstavlja stavka uklanja se iz `ToDoItems` niza,
5. dva HTML elementa kojima se stavka predstavlja unutar prezentacije uklanjaju se iz DOM strukture.

Markiranje stavki

Pod pojmom markiranja stavki podrazumeva se označavanje neke stavke kao kompletirane. Korisnik ToDo aplikacije ima mogućnost da ciklično promeni status stavke klikom na nju. Tako klikom na stavku ona postaje kompletirana ukoliko to nije bila i obrnuto. Takva logika se postiže korišćenjem sledeće metode:

```
function registerClickHandlers(toDoItemViewModel) {  
    for (let i = 0; i < toDoItemViewModel.views.length; i++) {  
  
        toDoItemViewModel.views[i].onclick = function (e) {  
            var id = this.dataset.id;  
            var index = toDoItems.findIndex(item => item.data.id === id);  
  
            toDoItemViewModel.data.completed =  
!toDoItemViewModel.data.completed;  
  
            if (toDoItemViewModel.data.completed) {  
  
                toDoItemViewModel.views[0].classList.add("completed");  
                toDoItemViewModel.views[1].classList.add("completed");  
  
                rootElementCompleted.appendChild(toDoItemViewModel.views[1]);  
  
            } else {  
                toDoItemViewModel.views[0].classList.remove("completed");  
                toDoItemViewModel.views[1].classList.remove("completed");  
  
                rootElementActive.appendChild(toDoItemViewModel.views[1]);  
            }  
        }  
    }  
}
```

Baš kao i unutar metode za preplatu na događaje brisanja, i ovde se sada obavlja registrovanje `click` događaja za sve DOM elemente kojima se predstavlja jedna ToDo stavka. Kada korisnik klikne na neku stavku, obavlja se sledeće:

1. dolazi se do vrednosti `data-id` atributa,
2. na osnovu vrednosti `data-id` atributa pronađi se objekat unutar niza kojim se predstavlja ToDo stavka na koju je korisnik kliknuo,
3. menja se vrednost `completed` svojstva `ToDoItem` objekta; njegova vrednost se inverte, odnosno ukoliko je bila `true`, postaje `false` i obrnuto,
4. ukoliko je stavka postala kompletirana, na dva HTML elementa koji je predstavlja postavlja se klasa `.completed`, koja kao efekat ima primenu `line-through` stilizacije; takođe, HTML element, koji je bio u listi aktivnih stavki, prebacuje se u listu kompletiranih,
5. ukoliko je stavka postala aktivna, sa dva HTML elementa koji predstavljaju ToDo stavku uklanja se klasa `.completed`, a HTML element kojim je stavka prikazana unutar liste kompletiranih stavki prebacuje se u listu aktivnih stavki.

Čuvanje podataka unutar Local Storagea

Kako bi se omogućilo čuvanje stavki i nakon zatvaranja prozora pregledača ili nakon osvežavanja stranice, kreiraćemo funkciju koja će čuvati stavke unutar skladišta Local Storage. Ideja je da se sve stavke čuvaju kao vrednost jednog ključa i to u JSON obliku.

Metoda za čuvanje stavki izgledaće ovako:

```
function saveToLocalStorage() {  
    localStorage.setItem('todo-data', JSON.stringify(todoItems));  
}
```

Logika metode za čuvanje stavki unutar skladišta Local Storage vrlo je jednostavna. Podaci se unutar Local Storage skladišta upisuju kao JSON tekst pod ključem todo-data. Prikazanom logikom se u JSON tekst pretvara niz todoItems kojim se predstavljaju objekti stavki zajedno sa referencama na pripadajuće HTML elemente kojima se stavke predstavljaju unutar prezentacije. Nama nije potrebno da čuvamo reference na HTML elemente, već samo čiste podatke. Stoga, ovakva logika sama po sebi nije dovoljna kako bi se obavilo ono što smo zamislili. Neophodno je definisati i proizvoljnu logiku kojom će da bude obavljena serijalizacija objekata niza todoItems. S obzirom na to da je reč o objektima koji se kreiraju korišćenjem konstruktorske funkcije ToDoItemViewModel(), unutar takve funkcije dovoljno je dodati metodu toJSON():

```
let ToDoItemViewModel = function(todoItem, views) {  
    this.data = todoItem;  
    this.views = views;  
  
    this.toJSON = function() {  
        return this.data;  
    }  
}
```

Metoda `toJSON()` aktiviraće se tokom serijalizacije i, umesto da isporuči kompletan objekat, zajedno sa DOM elementima, metoda `toJSON()` definiše da će serijalizacija podrazumevati samo data svojstvo.

Funkciju za čuvanje unutar Local Storagea potrebno je pozivati na nekoliko mesta:

- prilikom kreiranja novih stavki,
- prilikom brisanja stavki,
- prilikom promene statusa.

Čitanje podataka iz Local Storagea

Podatke upisane unutar Local Storagea neophodno je pročitati prilikom otvaranja aplikacije i, na osnovu podataka koji su sačuvani, kreirati stavke. Funkcija za čitanje podataka iz skladišta Local Storage i za njihovu integraciju nazad u aplikaciju izgleda ovako:

```
function loadFromLocalStorage() {  
    var json = localStorage.getItem('todo-data');  
  
    if (json === null)  
        return;  
  
    let ToDoItems = JSON.parse(json, (key, value) => {  
        if (key === "date") {  
            value = new Date(value);  
        }  
        return value;  
    });  
  
    if (ToDoItems.length === 0)  
        return;  
  
    for (let i = 0; i < ToDoItems.length; i++) {  
        generateToDoItemView(ToDoItems[i]);  
    }  
}
```

Podaci se čitaju pozivanjem `getItem()` metode kojoj se prosleđuje `todo-data` ključ. Ukoliko unutar Local Storagea ne postoji ključ `todo-data`, metoda `getItem()` kao svoju povratnu vrednost vraća `null`. Stoga se, odmah nakon čitanja, proverava vrednost promenljive `json`. Ukoliko je `json` `null`, napušta se metoda za čitanje podataka iz Local Storagea.

Ukoliko ključ `todo-data` postoji unutar Local Storagea, prelazi se na obradu podataka. S obzirom na to da su pročitani podaci u tekstualnom JSON obliku, neophodno ih je prevesti u JavaScript objekte. To se obavlja pozivanjem metode `parse()` `JSON` objekta. Na proces parsiranja se delimično utiče definisanjem reviver funkcije – podaci o datumu i vremenu se iz tekstualnog oblika prevode u `Date` objekte JavaScript jezika. Nakon obavljenog parsiranja, proverava se da li unutar dobijenog niza ima elemenata. Ukoliko elemenata nema, metoda se napušta pozivanjem naredbe `return`.

Na kraju, ukoliko postoje zapisi koji su pročitani i smeštani unutar `ToDoItems` niza, za svaki od dobijenih JavaScript objekata se poziva metoda `generateToDoItemView()`, koja obavlja generisanje prikaza i registrovanje nešto ranije prikazanih funkcija za obradu događaja.

Metodu za učitavanje ToDo stavki iz skladišta Local Storage potrebno je pozvati prilikom prvog pokretanja aplikacije.

Interfejs za korišćenje ToDo objekta

Do sada ste mogli da vidite da je kompletna logika ToDo aplikacije enkapsulirana unutar `ToDo()` konstruktorske funkcije kojoj se isporučuju reference na tri DOM objekta koji predstavljaju HTML elemente unutar kojih će biti prikazivane odgovarajuće stavke. Stoga, kako bi se koristila logika koja je definisana u prethodnim redovima, prvo je neophodno obaviti kreiranje objekta korišćenjem `ToDo()` konstruktorske funkcije:

```
var allItemsContainer = document.getElementById("all-items-container");
var activeItemsContainer = document.getElementById("active-items-
container");
var completedItemsContainer = document.getElementById("completed-items-
container");

var ToDoApp = new ToDo(allItemsContainer, activeItemsContainer,
completedItemsContainer);
```

Prikazanim kodom inicijalizuje se ToDo logika. Dolazi se do reference na tri HTML elementa za prikaz stavki, a zatim se tako dobijeni DOM objekti prosleđuju ToDo() konstruktorskoj funkciji.

Pored prosleđivanja elemenata za prikaz stavki, na korisniku ToDo objekta je još i da inicira proces upisivanja nove stavke. Upravo zbog toga, konstruktorska funkcija ToDo() unutar sebe poseduje jedan Closure, koji omogućava korisniku da inicira kreiranje nove stavke:

```
var ToDo = function(rootElementAll, rootElementActive, rootElementCompleted)
{
    ...
    return {
        add: function(content) {
            let ToDoItem = new ToDoItem(content, new Date());
            generateToDoItemView(ToDoItem);
            saveToLocalStorage();
        }
    }
}
```

Na ovaj način, korisnik ToDo objekta moći će da pristupi samo metodi add() i ni jednoj drugoj metodi, niti svojstvu koje se nalazi unutar takve konstruktorske funkcije. Metoda add() prihvata jedan parametar koji je prilikom poziva potrebno popuniti string vrednošću koju je korisnik uneo unutar polja za unos nove stavke:

```
var contentInput = document.getElementById("content");
var addButton = document.getElementById("add-btn");

addButton.onclick = function () {
    if (contentInput.value !== '') {
        ToDoApp.add(contentInput.value);
        contentInput.value = '';
    }
};
```

Prikazanim kodom obavlja se pretplata na `click` događaj na dugme za dodavanje nove stavke. Ukoliko je korisnik uneo neku vrednost u `input` polje za unos teksta stavke, poziva se nešto ranije prikazana `add()` metoda, `ToDo` objekta, čime se inicira proces kreiranja stavke, njena integracija unutar DOM strukture aplikacije i čuvanje unutar Local Storagea.

Zaključak

U prethodnim redovima ste mogli da vidite kako izgleda proces kreiranja jedne relativno jednostavne JavaScript aplikacije, korišćenjem Vanilla JS-a. Ono što ste mogli da primetite jeste da postoji zaista dosta aspekata o kojima je potrebno voditi računa, kako bi sve funkcionalisalo bez problema. Najveći izazov je svakako vođenje računa o vezi između podataka i njihove prezentacije unutar HTML strukture aplikacije. Jednostavno, u svakom trenutku je neophodno osigurati da podaci i njihova prezentacija budu sinhronizovani. Na primerima kompleksnijih aplikacija takvo nešto može predstavljati još veći izazov.

Drugi veliki problem kreiranja JavaScript aplikacija korišćenjem Vanilla JS-a jeste standardizacija unutrašnje strukture aplikacije koja se kreira. Mi smo, na primer, probleme prilikom kreiranja ToDo aplikacije rešili na načine prikazane u ovoj lekciji. Ipak, to ne znači da bi i neki drugi programer logiku ToDo aplikacije formulisao na identičan način. Takva činjenica prilikom timskog rada može proizvesti probleme neusklađenosti, koje je moguće prevazići samo uspostavljanjem jasnih konvencija i preciznim planiranjem aplikativne strukture.

Dva upravo navedena problema veoma lako se prevazilaze korišćenjem neke od popularnih biblioteka namenjenih frontend razvoju. Nakon priče o izvršnom okruženju Node.js i npm menadžeru paketa, ostatak ovoga kursa biće posvećen upoznavanju nekoliko takvih biblioteka. Na primeru razvoja ToDo aplikacije bićete u mogućnosti da direktno uporedite Vanilla JS pristup i korišćenje nekih od najpopularnijih frontend biblioteka.

Rezime

- *Vanilla JS* je pojam koji se koristi da označi čist JavaScript kod, bez upotrebe bilo kakvih biblioteka ili softverskih okvira.
- Pojam *Vanilla JS* izmislio je 2012. godine Erik Vastl (Eric Wastl), kao šalu koja treba da podseti frontend programere da je i bez upotrebe popularnih i modernih biblioteka i softverskih okvira moguće kreirati JavaScript aplikacije.
- Preduslov za korišćenje različitih JavaScript biblioteka je odlično poznavanje JavaScript jezika i Web API-ja.
- Rutiranje je pojam koji predstavlja usklađivanje odgovarajućeg resursa sa URL adresom.
- Prilikom korišćenja tradicionalnih, višestraničnih web sajtova, rutiranje obavlja web pregledač.
- Prilikom razvoja Single-page aplikacija, rutiranje se mora obaviti na klijentu, od JavaScript logike.
- Location API jeste naziv za skup funkcionalnosti web pregledača koji omogućava rukovanje URL adresama na klijentu.
- Objekat kojim se predstavlja URL adresa trenutno učitane web stranice je `Location`, a njemu je moguće pristupiti korišćenjem svojstava `document.location` i `window.location`.
- hashchange je događaj koji se aktivira kada se promeni identifikator fragmenta unutar URL adrese.