Progresivne web-aplikacije

Nekić, Maja

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Maja Nekić

PROGRESIVNE WEB-APLIKACIJE

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Zvonimir Bujanović

Zagreb, rujan, 2019.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Mami, tati i Andriji.
Bez vas ovo ne bi bilo moguće.
Hvala vam!

Sadržaj

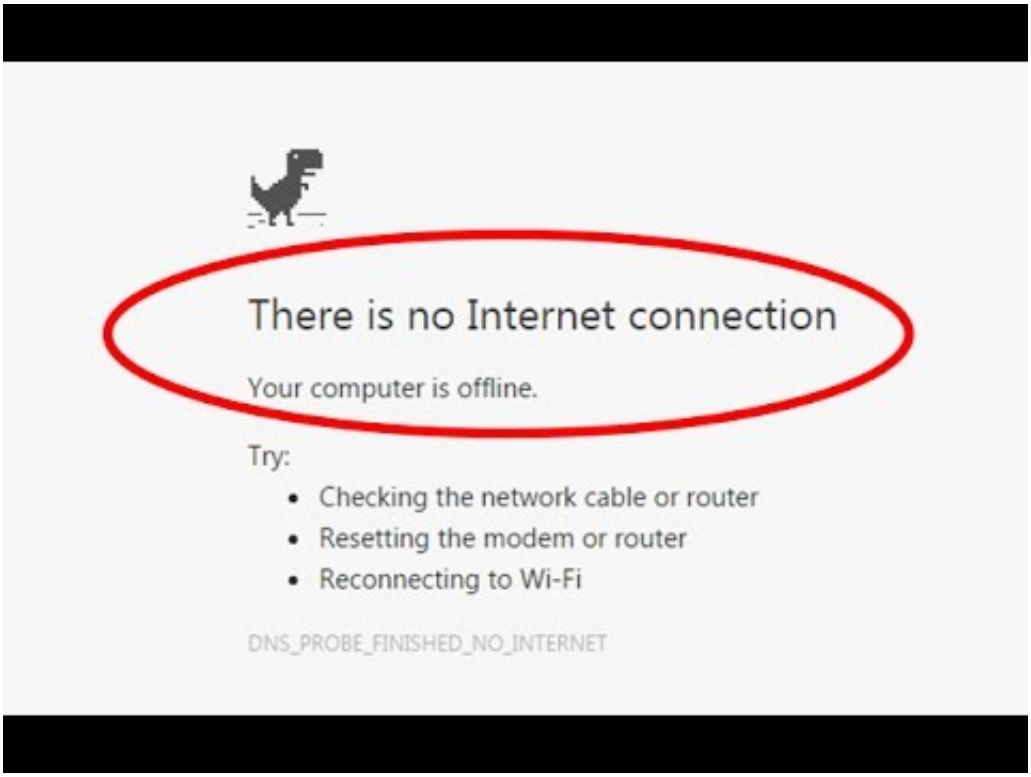
Uvod

Veza web-aplikacija i mobilnih uređaja započela je razvojem prvog iPhone-a 2007. godine koji je korisnicima omogućio pretraživanje web-sadržaja na mobilnom uređaju. Godinu dana kasnije pojavile su se mobilne aplikacije koje se moglo instalirati iz trgovine aplikacija (engl. app store) na mobilni uređaj. Značajkama poput napredne grafike, geolokacije, push obavijesti (engl. push notifications; vidi više u poglavlju 6) i još mnogo drugih, mobilne aplikacije su postale toliko popularne da su zasjenile web. Međutim, trenutno se nalazimo u razdoblju gdje je gotovo nemoguće zamisliti život bez web-aplikacija i pristupa internetu. Ako nas zanima recept nekog jela, nakon nekoliko klikova u našem web-pregledniku ćemo ga saznati. Ako nas zanima vozni red javnog prijevoza, također je preglednik tu da nam ga „servira“. Ako želimo saznati kakvo će vrijeme biti, gdje možemo kupiti jaknu za kišu koja dolazi ili u kojem frizerskom salonu možemo osušiti kosu koju je kiša smočila, odgovarajuća web-aplikacija će nam dati sve željene odgovore.

Sve to lijepo zvuči, no moramo biti svjesni činjenice da ćemo izgubiti pristup internetu svaki puta kada se vozimo u tunelu ili u liftu. Web-lokacija koju trenutno posjećujemo će nam prikazati poruku prikazanu na slici 0.1 koja kaže da nemamo internetsku vezu. Više nećemo moći vidjeti sadržaj koji smo gledali.

Naravno, internetska veza nije svuda iste jačine. Negdje je toliko slaba da će se željena web-aplikacija jedva otvarati. Otvaranje web-aplikacije može biti sporo i zbog same veličine sadržaja koji se nalazi na njoj. Ako dođemo u situaciju da nam se željeni sadržaj sporo učitava, vjerojatno ćemo tada otići s te web-lokacije i okušati svoju sreću na nekoj drugoj. Progresivna web-aplikacija nam ne bi zadavala toliko muke. Štoviše, ona nam može ponuditi puno više od običnih web-aplikacija na koje smo navikli. Progresivne web-aplikacije (PWA) su ono što su davne 2008. godine bile mobilne aplikacije – ogroman napredak mobilne tehnologije.

PWA su moderne web-aplikacije predviđene prvenstveno za mobilne uređaje, koje se mogu usporediti s ponajboljim mobilnim aplikacijama. One objedinjuju funkcionalnosti koje ostvaruju mobilne aplikacije i dohvat koji omogućuje web. Neke od najvažnijih odlika PWA su pouzdanost, privlačnost i brzina. Aplikacije se trenutno otvaraju te nikada ne prikazuju simbol prekida veze, čak i onda kada je internetska veza nestabilna.



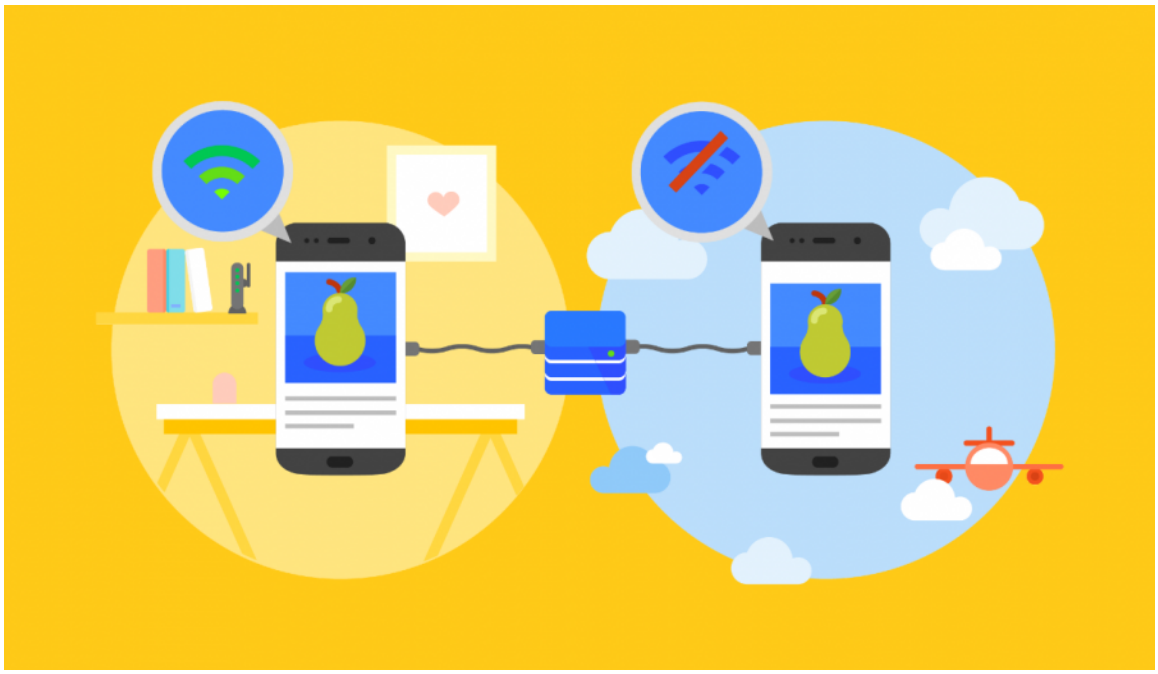
Slika 0.1: Poruka o prekidu internetske veze

Pokušajmo sada na primjeru ilustrirati PWA. Zanima nas vremenska prognoza za grad Zagreb. U web-preglednik na svom mobilnom uređaju vjerojatno upišemo nešto poput „vremenska prognoza Zagreb“. Kliknemo na neku od ponuđenih web-lokacija i saznamo da će narednih dana biti sunčano. Ista informacija nas zanima i sutradan, ponovno upisujemo ključne riječi u naš web-preglednik te odabiremo jednu od ponuđenih web-lokacija. Recimo da smo odabrali neku drugu web-lokaciju koja se učitava puno duže od one koju smo jučer posjetili. Nemamo vremena dugo čekati, izlazimo s te web-lokacije i ponovno odlazimo na onu koju smo jučer posjetili. Prognoza se nije promijenila, predviđa se sunčano vrijeme. No, uz vremensku prognozu, danas nam preglednik nudi mogućnost instaliranja aplikacije na naš mobilni uređaj. Klikom na odgovarajuću opciju, aplikacija se instalira i smješta na naš početni zaslon. Sljedeći put kad želimo saznati vremensku prognozu samo otvorimo aplikaciju na našem mobilnom uređaju. Također, aplikacija nas tada pita želimo li primati obavijesti o promjeni vremenske prognoze. Na to pitanje odogovorimo potvrdno, proučimo prognozu, vidimo da nas opet očekuje sunčano vrijeme i zatvorimo aplikaciju. Sljedeće jutro smo pripremili da ćemo za posao obući kratke rukave,

no aplikacija nam šalje obavijest da postoji mogućnost lokalnih pljuskova. U zadnji tren uzimamo kišobran i jaknu te izlazimo iz stana. Popodne je, kako nam je aplikacija i javila, počela padati kiša. Sva sreća da smo instalirali aplikaciju na mobilni uređaj.

Iz ovog malog primjera vidimo koliko prednosti progresivne web-aplikacije imaju ispred web-aplikacija. Analizirajmo prethodni primjer malo detaljnije. Web-preglednik nam uvijek ponudi pregršt opcija koje odgovaraju našim zahtjevima pretraživanja. Dakle, svaki puta možemo posjetiti drugu web-lokaciju da bismo saznali infomaciju koja nas zanima. Web-stranice žele privući i zadržati korisnike, a to može biti otežano jer, na primjer, čim se stranica učitava dulje nego što smo predvidjeli, mi je napuštamo i odabiremo neku drugu. Instalacijom aplikacije na mobilni uređaj, dotična web-lokacija osigurava da će korisnici ponovno koristiti tu aplikaciju (u suprotnom, zašto bi je uopće instalirali?!). Instalacija je jednostavnija nego ikada jer ne zahtijeva odlazak u trgovinu aplikacija, već se sve događa na web-stranici koju trenutno posjećujemo. Pružanjem tzv. push obavijesti aplikacija razvija interakciju s korisnicima te im javlja infomacije koje ih zanimaju. Jedna od važnijih činjenica jest ta da se aplikacija svaki puta trenutno otvara, bez obzira na stanje mreže. Sve navedeno osigurava da web-aplikacija zadrži svoje (zadovoljne) korisnike.

Na slici 0.2[1] vidimo da, u slučaju nestabilne veze, PWA prikazuje isti sadržaj.



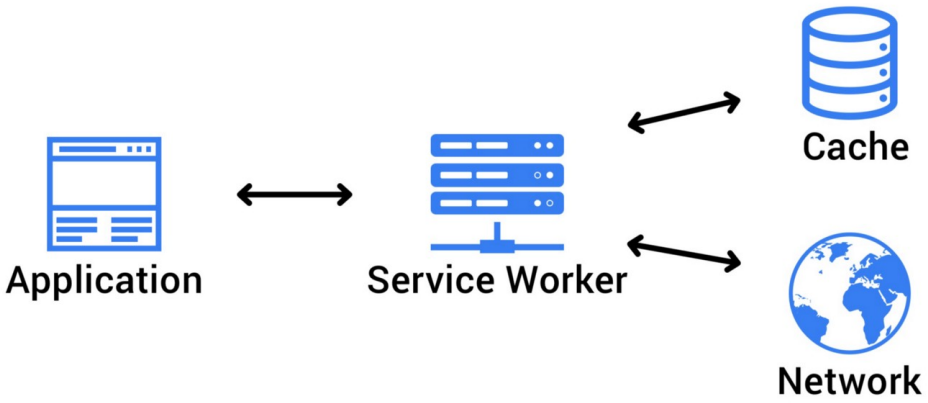
Slika 0.2: PWA prikazuje isti sadržaj i bez internetske veze

[1] Slika je preuzeta s [3].

Temeljni dio svake PWA je uslužna skripta. Prije uslužnih skripti, kôd web-aplikacija se izvodio ili na poslužitelju ili u prozoru preglednika. Uslužna skripta je skripta koja se može registrirati za upravljanje jednom ili više stranica naše web-lokacije. Jednom kada je instalirana, uslužna skripta se smješta izvan svakog prozora ili kartice preglednika. Dakle, postoji sloj između web-poslužitelja i web-stranice koji odgovara na zahtjeve neovisno o mrežnoj vezi.

Uslužna skripta može osluškivati i djelovati na događaje sa svih stranica koje su pod njezinom kontrolom. Ona otkriva izvanmrežno stanje ili spori odgovor poslužitelja te vraća sadržaj iz predmemorije (engl. cache). Također, nakon zatvaranja aplikacije, uslužna skripta i dalje komunicira s poslužiteljem te je zadužena za pružanje push obavijesti i osigurava da sve radnje koje je korisnik napravio budu dostavljene poslužitelju (na primjer, spremanje podataka u bazu podataka i slično).

Na slici 0.3[2]možemo vidjeti povezanost uslužne skripte s pojedinim dijelovima koji tvore PWA. Točnije, vidljivo je kako su uslužne skripte povezane s aplikacijom, predmemorijom te mrežom.



Slika 0.3: Uslužna skripta

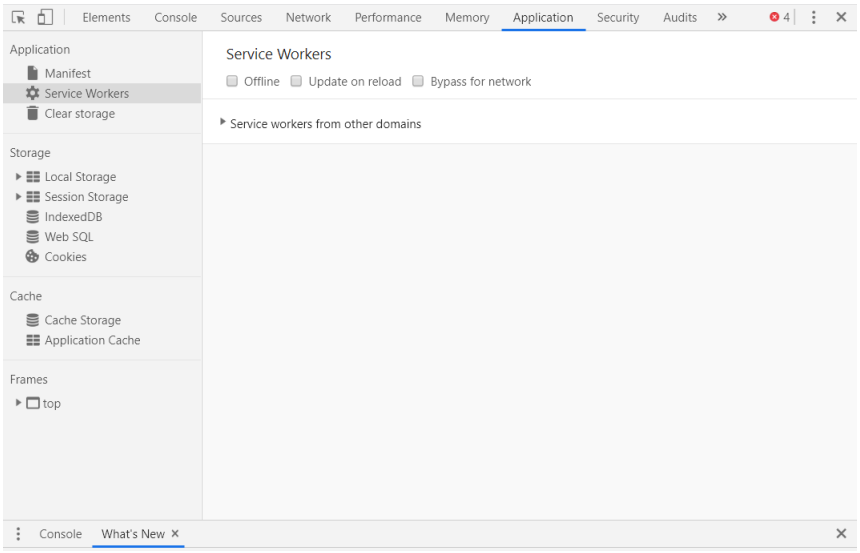
[2]Slika je preuzeta s [5].

Poglavlje 1

Polazna web-aplikacija i alati

1.1 Alati

U ovom radu slijedimo [1], [3], [5] te [4] kako bismo demonstrirali kako web-aplikaciju preoblikovati tako da ona postane progresivna. Svaki korak u toj preobrazbi ćemo detaljno prikazati i objasniti te potkrijepiti informacijama koje su važne za dotični dio s teorijskog stajališta. Objasnit ćemo ulogu manifesta te detaljno obraditi uslužne skripte na koje je već dan osvrt. Upoznat ćemo se s lokalnom bazom podataka, točnije s IndexedDB (vidi više u poglavlju 4), obavijestima, push obavijestima te pozadinskom sinkronizacijom (engl. background sync; vidi više u poglavlju 5).



Slika 1.1: Alati za razvojne programere

Za razvoj PWA koristimo Google Chrome jer ima podršku za sve značajke koje želimo implementirati. Kao što je vidljivo na slici 1.1, alati za razvojne programere (Ctrl + Shift + I) sadrže tri kartice koje će nam biti od velike pomoći. Radi se o Console, Network i Application. Sve pomoćne poruke koje ispisujemo tijekom razvijanja aplikacije biti će prikazane u Console i to nam uvelike olakšava pronalaženje grešaka. Komunikaciju klijentskih i poslužiteljskih skripti možemo pratiti u Network-u (zahtijeve klijenta i odgovore poslužitelja). U Application se nalaze kartice potrebne za razvoj PWA od kojih ćemo se mi koristiti sljedećima: Manifest, Service Workers, Session Storage, IndexedDB te Cache Storage. Service Workers nam nudi tri mogućnosti: *Offline, Update on reload, Bypass for network*. Kada želimo vidjeti kako naša aplikacija radi bez internetske veze, označit ćemo *Offline*. Također, kako bismo bili sigurni da uvijek koristimo ažurnu verziju uslužne skripte, prilikom razvoja PWA cijelo vrijeme ćemo imati označenu opciju *Update on reload*.

1.2 Početna aplikacija

Aplikacija od koje ćemo krenuti je web-aplikacija namjenjena studentima za praćenje rezultata na ispitima. Web-aplikacija se sastoji od klijentskog i poslužiteljskog dijela. Klijentski dio je pisan u jeziku JavaScript, dok je poslužiteljski pisan u jeziku PHP. Oni komuniciraju pomoću Ajax upita koristeći metodu *GET*. Ajax upiti označavaju asinkrone upite prema poslužiteljskoj skripti koje generira klijentski dio aplikacije. Oni ne zahtijevaju ponovno učitavanje web-stranice, već ju klijentski dio osvježi novim podacima kada dobije odgovor poslužiteljske skripte. Pri svakom Ajax upitu koji je došao od strane klijenta, poslužitelj se spaja na bazu *Studenti* te izvodi potrebe radnje. Kao odgovor se šalje ili prikladna poruka (na primjer, poruka o razlogu zbog kojeg prijava studenta u aplikaciju nije uspjela) ili traženi podaci (na primjer, identifikacijski broj studenta).

Klijentski dio web-aplikacije sastoji se od skripti koje se prikazuju korisnicima. Dakle, to su skripte kojima je implementirana početna stranica za prijavu (*index.html*), stranica na kojoj se studentima prikazuju rezultati (*rezultati.html*), početna stranica koja se prikazuje administratoru (*administrator.html*) i slično.

Poslužiteljska strana sastoji se od skripti implementiranih za rad s bazom. Neke od njih su: *rezultati.php*, *index.php* i ostale. Iz navedenog je vidljivo da skripte klijentske strane imaju ekstenziju *.html*, a one poslužiteljske strane *.php*. Kako svaka skripta "servirana" klijentu zahtijeva neke podatke iz baze, očito uvijek mora postojati odgovarajuća poslužiteljska skripta koja iste podatke dohvaća.

Sve varijable važne za uredno funcioniranje web-aplikacije se spremaju u Session Storage. Prilikom svake uspješne prijave, identifikacijski broj korisnika se pohranjuje u spomenuti spremnik. Ako je korisnik student, njegov identifikacijski broj jedak je atributu

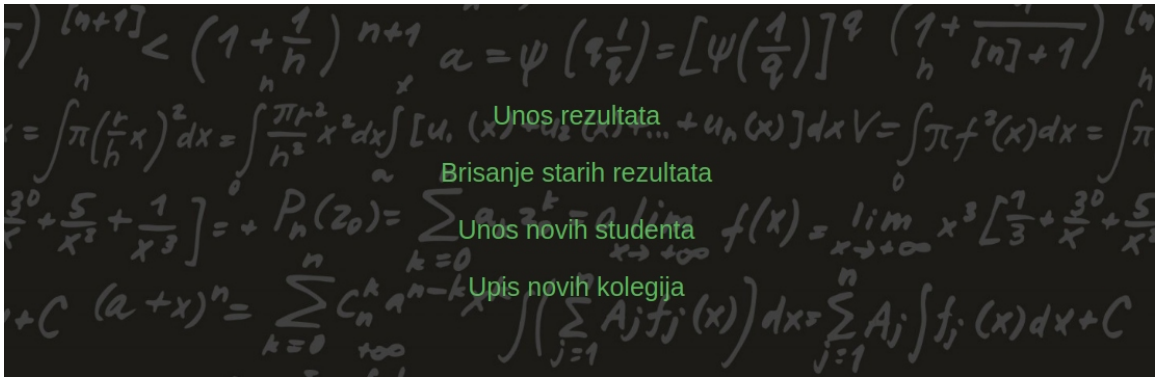
student_id iz tablice *studenti*. S druge strane, ako je korisnik administrator, u Session Storage se upisuje varijabla s vrijednosti 0.

Korisnici imaju mogućnost odjaviti se iz aplikacije. Ukoliko se odluče na odjavu, svi podaci koji su spremljeni u Session Storage se brišu. Posebno, briše se i varijabla koja označava identifikacijski broj korisnika. Ako korisnici napuste aplikaciju bez odjave, podaci u Session Storage-u ostaju spremljeni te, na temelju vrijednosti varijable u koju je upisan identifikacijski broj, aplikacija razlikuje studente od administratora.

Razlikovati „vrstu“ korisnika je važno zato što nakon otvaranja aplikacije sadržaj koji treba biti vidljiv studentima nije jednak sadržaju koji se prikazuje administratoru. Na slikama 1.2 i 1.3 se mogu uočiti spomenute razlike.

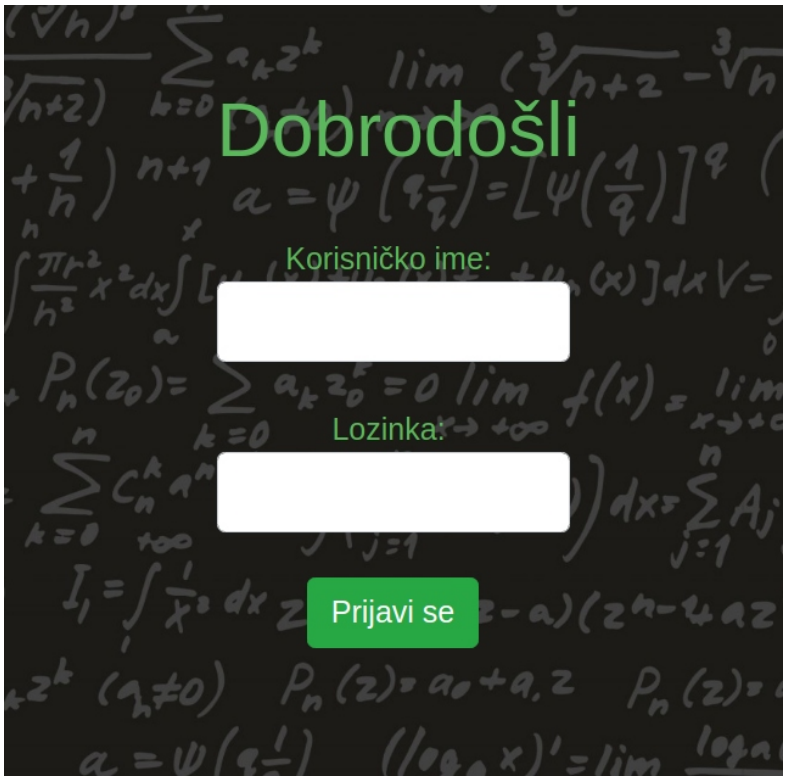
Kolegij	Matematicka analiza 1	Matematicka analiza 2
1.kolokvij	25	25
2.kolokvij	13	13
Završni ispit	5	-
1.zadaća	17	-
2.zadaća	15	-
3.zadaća	35	-
4.zadaća	1	-

Slika 1.2: Sadržaj vidljiv studentima u aplikaciji



Slika 1.3: Početna stranica vidljiva administratoru u aplikaciji

Prilikom upisa prve akademske godine na željenom fakultetu svakom studentu se dodijeli jedinstveni matični broj akademskog građana (JMBAG) te lozinka.



Slika 1.4: Prijava u aplikaciju

Na slici 1.4 vidimo da se prilikom prijave zahtijevaju korisničko ime te lozinka. Kao korisničko ime studenti koriste JMBAG, a kao lozinku koriste lozinku koja im je dodijeljena.

Slijedi komunikacija klijenta i poslužitelja u kojoj poslužitelj provjerava točnost podataka koje mu je klijent poslao. Ako je sve u redu, prijava je uspjela te poslužitelj kao odgovor šalje rezultate i identifikacijski broj korisnika koji se prijavio. Identifikacijski broj se sprema u Session Storage. Studentu se sada prikaže popis svih upisanih kolegija te bodovi iz pojedinih elemenata ocjenjivanja. Inače, ako poslužitelj nije uspio u bazi pronaći odgovarajući par podataka, kao odgovor šalje poruku koja sadrži informacije o greški. Poruka se prikazuje korisniku kao upozorenje (na primjer, nepostojeće korisničko ime, pogrešna lozinka ili slično). Studenti imaju na izbor žele li se odjaviti prije napuštanja web-aplikacije ili žele ostati prijavljeni u istu. Ako se odjave, podaci iz Session Storage-a se brišu (obriše se identifikacijski broj studenta koji se prijavio) te je sljedeći put prilikom otvaranja web-aplikacije potreba prijava. S druge strane, ako se ne odjave, prilikom otvaranja web-aplikacija za spremljeni identifikacijski broj iz Session Storage-a dohvati od servera rezultate. Studenti se automatski preusmjeravaju na stranicu s njihovim rezultatima.

Baza podataka korištena za ovu web-aplikaciju naziva se *Studenti*. *Studenti* se sastoji od 3 glavne tablice: *studenti*, *kolegiji* te *rezultati*. Tablica *studenti* sadrži sve važne podatke o studentima te ima sljedeće atribute: *student_id* (primarni ključ), *username*, *password*, *ime*, *prezime*. Tablica *kolegiji* se sastoji od popisa kolegija koji su dostupni na fakultetu te sadrži atribute *kolegij_id* (primarni ključ) i *naziv_kolegija*. Vidimo da su *student_id* i *kolegij_id* jedinstveni identifikatori za svakog studenta, odnosno za svaki kolegij. Posljednja tablica, *rezultati*, se sastoji od parova id-a studenta i id-a kolegija te broja bodova za svaki od postojećih elemenata ocjenjivanja. Dakle, atributi su: *studenti_id*, *kolegij_id*, *1kolokvij*, *2kolokvij*, *zavrsni*, *1zadaca*, *2zadaca*, *3zadaca* i *4zadaca*. Navedena baza se kreira pokretanjem skripte *prepareDB.php*.

Administrator ima posebno korisničko ime i lozinku te, za razliku od studenata, on ima na raspolaganju druge funkcionalnosti. Sve funkcionalnosti navodimo u nastavku i dajemo za svaku kratki opis.

1. Upis bodova: poslužitelj klijentu šalje popis svih kolegija te administrator može odabrati za koji kolegij želi studentima upisati nove bodove. Odabirom naziva kolegija, Ajax upitom se dohvaća popis svih studenata i elemenata ocjenjivanja, s već postojećim rezultatima. Isti popis se prikazuje te se pruža mogućnost upisa novih bodova. Reakcija na događaj promjene sadržaja *input* polja Ajax upitom šalje sve potrebne podatke poslužitelju koje isti sprema u bazu podataka i time ažurira tablicu rezultati.

2. Brisanje upisanih kolegija: Ajax-om se dohavati popis svih studenata te se prikaže

administratoru. On odabire studenta kojemu iz baze želi obrisati kolegije upisane prethodnih akademskih godina. Odabirom studenta, Ajax-om se dobiva popis kolegija dotičnog studenta. Nakon odabira željenog predmeta, poslužiteljskoj strani aplikacije se šalju potrebni podaci te ona ažurira bazu podataka, točnije tablicu *rezultati* tako da obriše jedan redak te tablice (onaj koji je jednoznačno određen poslanim podacima).

3. Unos novih studenata: na početku svake akademske godine administrator upisuje studente koji su upisali prvu godinu fakulteta u bazu. Klijentska strana poslužiteljskoj šalje podatke o studentu koji se upisuju u bazi podataka u tablicu *studenti*.

4. Upis novih kolegija: administrator ima mogućnost upisivati nove kolegije studentima koji se nalaze u bazi. Odabirom te opcije, poslužiteljska strana klijentskoj šalje popis svih studenata. Administrator s tog popisa odabire studenta kojemu želi upisati nove kolegije i podaci o odabiru studenta se šalju poslužitelju. Nakon toga kao odgovor stiže popis svih kolegija te administrator odabire koje kolegije će upisat dotičnom studentu. Označavanjem kolegija, automatski se u tablicu *rezultati* dodaje novi redak s odgovarajućim podacima. Odznačavanjem već označenog kolegija iz tablice *rezultati* se briše redak koji ima odgovarajuće podatke.

Poglavlje 2

Uslužne skripte

2.1 Životni ciklus uslužne skripte

Za pravilno korištenje uslužnih skripti važno je razumjeti njihov životni ciklus koji se, tipično, sastoji od tri važne faze: registracije, instalacije i aktivacije.

Registracija

Uslužne skripte su skripte pisane u jeziku JavaScript koje reagiraju na događaje, a rade u pozadini aplikacije kao dodatni komunikacijski sloj između mreže (web-poslužitelja) i web-aplikacije. U svom pozadinskom radu, one presreću mrežne zahtjeve te trajno pohranjuju (predmemoriraju) podatke. Na taj način omogućen je izvanmrežni rad. Prije nego što možemo početi s korištenjem uslužne skripte, moramo je registrirati kao pozadinski proces. Kako bismo napravili svoju prvu uslužnu skriptu, potrebne su dvije nove skripte unutar naše aplikacije. U korijenski direktorij smjestimo skriptu *sw.js*, a skriptu *app.js* možemo smjestiti bilo gdje unutar direktorija. Važno je da je uslužna skripta smještena u korijenskom direktoriju jer ona kontrolira sav mrežni promet unutar aplikacije.

Datoteka *app.js* se sastoji od sljedećeg koda koji je namijenjen registraciji uslužne skripte.

```
if("serviceWorker" in navigator) {
  navigator.serviceWorker.register("sw.js").then(function(registration) {
    console.log("SW registered with scope:", registration.scope);
  }).catch(function(err) {
    console.log("Service Worker registration failed:", err);
  });
}
```

Započinje se provjerom podržava li preglednik uslužne skripte. Ako podržava, uslužna skripta se registrira i navedeni dio koda vraća *promise* koje se rješava nakon uspješne registracije uslužne skripte.

Skriptu *app.js* potrebno je izvršavati pri svakom otvaranju bilo koje druge klijentske skripte (one s ekstenzijom *.html*). U tu svrhu unutar HTML *tag-a head*, dodajemo

```
<script type="text/javascript" src="app.js"></script>
```

kako bismo osigurali da se skripta *app.js* izvrši prije svega ostalog na trenutnoj skripti.

Instalacija

Činjenica da je uslužna skripta registrirana ne znači da je ona i instalirana u našoj aplikaciji. Zbog toga nakon registracije slijedi faza instalacije uslužne skripte. Nakon uspješne registracije, skripta se preuzima i preglednik je pokušava instalirati pomoću sljedećeg koda koji se nalazi u datoteci *sw.js*.

```
self.addEventListener("install", function(event) {
  console.log('Service Worker installed');
});
```

Instalacija će se dogoditi samo ako uslužna skripta još nije registrirana (za svaku uslužnu skriptu, instalacija se događa samo jednom).

Kako uslužne skripte reagiraju na događaje, jasno je da se u *sw.js* skriptu dodaje novi događaj *install* koji pokreće instalaciju. Pomoću ovog događaja možemo obaviti neke zadatke specifične za aplikaciju (na primjer, predmemoriranje svih sadržaja koje želimo spremiti u predmemoriju te ih koristiti čak i onda kada nemamo internetsku vezu). No, više o tome malo kasnije.

Aktivacija

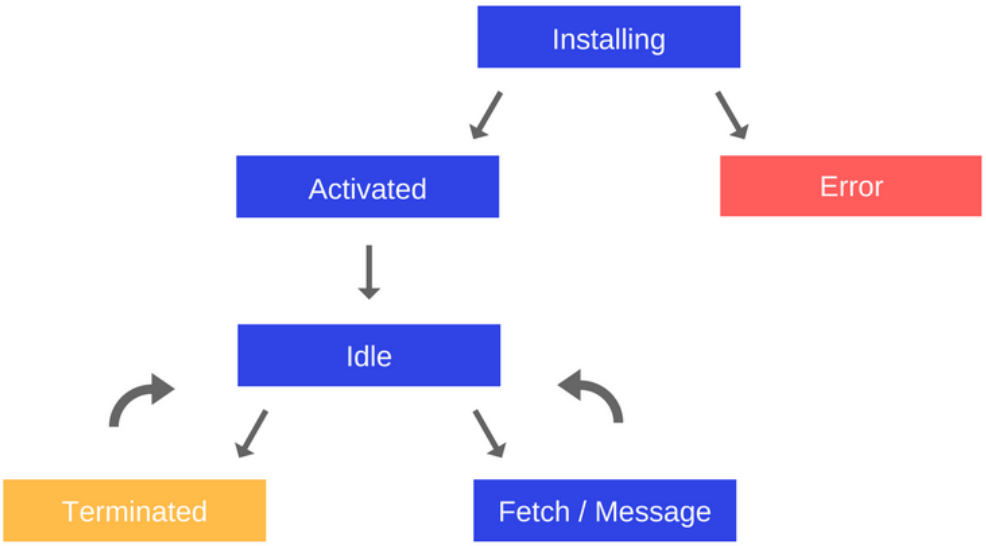
Ako je instalacija uspjela, uslužna skripta ulazi u stanje *installed* tijekom kojeg čeka na preuzimanje kontrole nad aplikacijom od trenutne uslužne skripte. Na red dolazi faza aktivacije uslužne skripte.

Uslužna skripta se ne aktivira odmah nakon instalacije. Do aktivacije će doći ako u datoteku *sw.js* dodamo te pokrenemo kôd u nastavku i to samo ako trenutno nema aktivne uslužne skripte ili ako korisnik osvježi web-stranicu.

```
self.addEventListener("activate", function(event) {
  console.log('Service Worker activated');
});
```

Poput događaja instalacije, i događaj aktivacije može u sebi sadržavati neke specifične radnje vezane uz aplikaciju (na primjer, možemo obrisati sadržaj predmemorije). Jednom kada je uslužna skripta aktivirana, ona ima potpunu kontrolu nad aplikacijom te može upravljati raznim događajima poput *fetch*, *push* i *sync*.

U slučaju da aktivna uslužna skripta ne primi niti jedan od gore navedenih događaja, ona prelazi u stanje mirovanja (engl. idle state). Nakon što je neko vrijeme u stanju mirovanja, skripta prelazi u prekinuto stanje (engl. terminated state). Prekinuto stanje ne znači da je uslužna skripta deinstalirana ili odjavljena (engl. unregistrated). Štoviše, ona će ponovno prijeći u stanje mirovanja čim krene primati funkcijske događaje (*fetch*, *push* ili *sync*). Opisana stanja uslužne skripte ilustrirana su slikom 2.1[1].



Slika 2.1: Životni ciklus uslužne skripte

[1] Slika je preuzeta s [3].

2.2 HTTPS i uslužne skripte

S obzirom na to da uslužne skripte presreću sve mrežne zahtjeve, uvijek postoji mogućnost prisustva zlonamjerne treće strane koja bi mogla oštetiti našeg korisnika. Upravo zbog toga samo stranice koje se poslužuju preko sigurnih veza (HTTPS) mogu registrirati uslužne skripte. Tijekom razvoja PWA, uslužne skripte možemo koristiti na našem lokalnom poslužitelju (localhost), ali nakon implementacije web-aplikaciju je potrebno postaviti na poslužitelj koji poslužuje preko sigurne veze HTTPS.

Poglavlje 3

Predmemoriranje

Kako bismo omogućili rad web-aplikacije čak i u uvjetima gdje nema internetske veze, potrebno je predmemoriranje sadržaja. Za predmemoriranje koristimo *CacheStorage API*.

CacheStorage je novi sloj predmemoriranja koji u potpunosti možemo kontrolirati. Možemo odlučiti što želimo spremiti u predmemoriju, što i kada želimo obrisati iz iste te na koji način odgovaramo zahtjevima iz preglednika (vraćajući traženo iz predmemorije ili s mreže).

Postavlja se pitanje kada predmemorirati sadržaj. Već je spomenuto da se ta radnja može obaviti tijekom instalacije uslužne skripte, to jest, unutar događaja *install*. Kako se taj događaj dogodi samo jednom u životnom ciklusu uslužne skripte, i naše predmemoriranje će se obaviti samo jednom. Također, ako nešto s predmemoriranjem pođe po krivu, uvijek imamo mogućnost otkazati instalaciju uslužne skripte. Ako dođe do prekida instalacije, preglednik će ponovno pokušati pokrenuti instalaciju sljedeći put kada korisnik posjeti stranicu. Predmemoriranjem unutar događaja *install* osigurali smo da se pohrane sve potrebne datoteke prije nego što se smatra da je uslužna skripta instalirana i aktivna.

3.1 Uobičajeni obrasci spremanja

Različiti obrasci spremanja odgovaraju različitim situacijama i zahtjevima. Primjerice, ako radimo aplikaciju za prognozu vremena, logično je da želimo koristiti obrazac koji uvijek prvo pokuša dohvatiti najnovije informacije o vremenu s mreže, a samo ako to dohvaćanje ne uspije, poseže se za već spremljenim informacijama u predmomoriju.

Uslužne skripte presreću sve zahtjeve za resursima pomoću događaja *fetch*. Dakle, kada preglednik zatraži neki resurs, uslužna skripta, osluškivajući događaje *fetch*, odgovara traženim resursom (ako takav postoji).

U nastavku slijedi nekoliko uobičajenih obrazaca predmemoriranja, a svi se oni implementiraju unutar datoteke *sw.js*.

Samo predmemorija

Na sve zatjeve za resursima se odgovara datotekama iz predmemorije. Ako traženi resurs nije pronađen u predmemoriji, zahtjev ne uspijeva. Ovaj obrazac pretpostavlja da je resurs pohranjen u predmemoriju nekad prije (vjerojatno tijekom instalacije uslužne skripte).

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    caches.match(event.request)
  );
});
```

Obrazac je pogodan za statičke resurse koji se ne mijenjaju često. Primjerice, ikone, logotipe i slično.

Predmemorija, posezanje na mrežu

Slično kao i gore, ovaj obrazac odgovara zahtjevima sadržajem iz predmemorije. No, ako traženi sadržaj nije pronađen, uslužna skripta ga pokušava pronaći na mreži.

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request)
    })
  );
});
```

Samo mreža

Ovo je klasični model weba gdje se zahtjevi pokušavaju dohvatiti s mreže. Ako dohvaćanje ne uspije, zahtjev ne uspijeva. Obrazac je dobar za sadržaje koje za koje odlučimo da ih ne želimo predmemorirati.

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request)
  );
});
```

Mreža, posezanje u predmemoriju

Zahtjevi se uvijek pokušaju dohvatiti s mreže. Ako dohvaćanje ne uspije, traženi resurs se pokuša pronaći u predmemoriji. Ako resurs nije pronađen ni tada, zahtjev ne uspijeva.

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request);
    })
  );
});
```

Uz sve spomenute obrasce, postoje i razne varijante koje uključuju modifikacije istih. Slijede neke mogućnosti.

Predmemorija, posezanje na mrežu sa čestim ažuriranjima

Za resurse koji se mijenjaju s vremena na vrijeme, ali je prikazivanje njihove najnovije verzije manje važno od brzine samog odgovora, možemo modificirati obrazac *predmemorija, posezanje na mrežu*. Brzi odgovor na zahtjev za resurs dobiva se iz predmemorije, a u pozadini se poseže na mrežu kako bi se dohvatila najnovija verzija resursa i pohranila u predmemoriju. Sve promjene resursa dohvaćenog s mreže biti će dostupne sljedeći put kada korisnik zatraži taj resurs.

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    caches.open("cache-name").then(function(cache) {
      return cache.match(event.request).then(function(cachedResponse) {
        var fetchPromise = fetch(event.request)
                           .then(function(networkResponse) {
          cache.put(event.request, networkResponse.clone());
          return networkResponse;
        });
        return cachedResponse || fetchPromise;
      })
    })
  );
});
```

Mreža, posezanje u predmemoriju sa čestim ažuriranjima

Kada je važno uvijek dohvatiti najnovije verzije dostupnih resursa, modificiramo obrazac *mreža, posezanje u predmemoriju*. Ovaj obrazac uvijek pokušava dohvatiti najnoviju verziju traženog resursa s mreže te, ako ne uspije, u predmemoriji potraži spremljenu verziju. Dodatno, svaki puta kada se resurs uspješno dohvati s mreže, predmemorija se ažurira s mrežnim odgovorom.

```
self.addEventListener("fetch", function(event) {
  event.respondWith(
    caches.open("cache-name").then(function(cache) {
      return fetch(event.request).then(function(networkResponse) {
        cache.put(event.request, networkResponse.clone());
        return networkResponse;
      }).catch(function() {
        return caches.match(event.request);
      });
    })
  );
});
```

3.2 Planiranje strategije za predmemoriranje u našoj aplikaciji

Želimo li omogućiti svojim korisnicima izvanmrežni rad, potrebno je predmemorirati sve resurse. Kako bismo korisnicima pružili najbolju moguću uslugu, trebamo analizirati od kakvih se sve resursa sastoji naša web-aplikacija te, u skladu s time, odabrati najpogodnije obrasce za predmemoriranje istih.

Sve stranice naše aplikacije imaju statičke dijelove – slike, datoteke i poveznice koje određuju izgled aplikacije. Kako se navedeni dijelovi rijetko mijenjaju (ili čak nikada), za njihovo predmemoriranje koristimo obrazac *predmemorija, posezanje na mrežu*.

Što se tiče ostatka sadržaja aplikacije, većinom su to datoteke (skripte) koje se ne mijenjaju često između različitih verzija. Zaključujemo da bismo kôd njihovog predmemoriranja mogli koristiti isti obrazac. No, u ovom trenutku nam se javlja veliki problem. Ako se datoteke promijene, morat ćemo promijeniti i uslužnu skriptu kako bismo osigurali da se našim korisnicima prikazuje najnovija verzija stranice. Razmatramo li slučaj dalje, stvari postaju još gore. Naime, sve dok prva uslužna skripta ne prepusti kontrolu drugoj (novoj), korisnicima će se prikazivati stara verzija stranice. Druga uslužna skripta će preuzeti kontrolu nad stranicom tek kada korisnik drugi puta posjeti našu aplikaciju. Upravo zbog toga,

odbacujemo obrazac *predmemorija, posezanje na mrežu* te imamo sljedeća dva moguća slučaja:

1. Posluživanje datoteka obrascem *mreža, posezanje u predmemoriju*. Korisnicima će se uvijek prikazivati najnovije verzije stranica, no propuštamo priliku da ubrzamo učitavanje stranice dohvaćanjem iste iz predmemorije, ako ona tamo postoji.

2. Posluživanje datoteka obrascem *predmemorija, posezanje na mrežu sa čestim ažuriranjima*. Ova opcija uvijek poslužuje datoteke iz predmemorije, no istovremeno provjerava postoji li novija verzija istih te ažurira predmemoriju ako postoji. Dakle, dobivamo vrlo brzi odgovor, ali korisnicima se možda neće učitati najnovija verzija stranice. Doduše, to nije toliko veliki problem jer će sljedećim učitavanjem biti prikazana najnovija verzija (nije potrebno ažurirati uslužnu skriptu).

Aplikacija uvijek mora prikazati najnoviju verziju stranice *rezultati.html*. Dakle, puno važnija je točnost podataka nego brzina. Stoga se odlučujemo za prvu opciju posluživanja. Iz razloga što se ostale datoteke rijetko mijenjaju, njih poslužujemo pomoću druge opcije.

3.3 Implementacija uslužne skripte

Do sada smo u našu aplikaciju dodali implementiranu skriptu *app.js*, a *sw.js* smo samo kreirali. Sada ćemo toj skripti dodati potrebni kôd.

Prvo trebamo navesti sve datoteke koje želimo predmemorirati i dodati događaj *install* unutar kojega se odvija predmemoriranje.

```
var CACHE_NAME = "PWA-cache";
var CACHED_URLS = [
  "index.html",
  "view.css",
  "6.jpg",
  "https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css",
  "app.js",
  "administrator.html",
  //...
];

self.addEventListener("install", function(event) {
  console.log( 'sw.install' );
  event.waitUntil(
    caches.open(CACHE_NAME).then(function(cache) {
```

```
      return cache.addAll(CACHED_URLS);
    })
  )
});
```

Varijabla *CACHED_URLS* ne sadrži cijeli popis datoteka jer je poprilično dug pa je navedena samo nekolicina radi ilustracije.

Zbog toga što dohvaćamo i pohranjujemo datoteke asinkrono, želimo odgoditi događaj *install* sve dok asinkroni događaj *fetch* nije završio. Funkcija *waitUntil* nam to omogućava. Sljedećim pseudokodom ćemo opisati što zapravo ta funkcija radi.

```
ako je otkriven dogadaj instalacije,
nemoj ga proglasiti uspjesnim sve dok:
  uspjesno otvori predmemoriju
    onda
  uspjesno dohvati datoteke i spremi ih u predmemoriju
ako neki od ovih koraka ne uspije,
prekini instalaciju usluzne skripte
```

Funkcija *waitUntil* produlji trajanje događaja *install* sve dok se *promise* koji smo proslijedili ne razriješi. To nam omogućava da čekamo sve dok uspješno ne pohranimo datoteke u predmemoriju prije proglašenja događaja *install* završenim. Također, dobivamo mogućnost prekinuti instalaciju ako je u bilo kojem koraku došlo do odbijanja *promise-a*. Unutar *waitUntil* pozivamo *caches.open* i proslijeđujemo ime naše predmemorije. Funkcija *caches.open* ili otvori i vrati postojeću predmemoriju, ili, ako predmemorija s danim imenom ne postoji, stvara predmemoriju danog imena i vraća je. Objekt koji se vraća je ponovno *promise* te zato nastavljamo s *then* izjavom.

Nakon što se uslužna skripta instalira te je spremna postati aktivna i zamijeniti staru aktivnu skriptu, slijedi događaj *activate* koji uzrokuje aktivaciju iste te skripte. U tom trenutku, sve datoteke koje se zahtijevaju su uspješno pohranjene u predmemoriju. No, prije nego što proglasimo uslužnu skriptu aktivnom, želimo obrisati stare predmemorije koje su koristile stare uslužne skripte.

```
self.addEventListener("activate", function(event) {
  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.map(function(cacheName) {
          if(CACHE_NAME !== cacheName &&
                            cacheName.startsWith("PWA-cache")) {
```

```
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

Kôd započinje s produljivanjem događaja pomoću funkcije *waitUntil*. To znači da, kako bi završila svoju aktivaciju, uslužna skripta mora čekati sve dok ne obrišemo sve stare predmemorije. Tu radnju napravimo proslijeđujući *promise* funkciji *waitUntil*. Pomoću *caches.keys()* dobijemo niz koji sadrži imena svih predmemorija stvorenih u našoj aplikaciji. Iteracijom po istom brišemo stare predmemorije.

Razmotrili smo razne strategije predmemoriranja i dohvaćanja datoteka. Na redu je implementacija istog. Također, zbog duljine koda, navodimo samo dio. Ostatak je analogan.

```
self.addEventListener("fetch", function (event) {
  var requestURL = new URL(event.request.url);
  if (requestURL.pathname === "/~manekic/pwa/index.html" ) {
    event.respondWith(
      caches.open(CACHE_NAME).then(function(cache) {
        return cache.match("index.html")
                    .then(function(cachedResponse) {
          var fetchPromise = fetch("index.html")
                               .then(function(networkResponse) {
            cache.put("index.html", networkResponse.clone());
            return networkResponse;
          });
          return cachedResponse || fetchPromise;
        });
      })
    );
  } else if (requestURL.pathname === "/~manekic/pwa/administrator.html") {
    event.respondWith(
      caches.open(CACHE_NAME).then(function(cache) {
        return cache.match("administrator.html")
                    .then(function(cachedResponse) {
          var fetchPromise = fetch("administrator.html")
                               .then(function(networkResponse) {
            cache.put("administrator.html", networkResponse.clone());
```

```
            return networkResponse;
          });
          return cachedResponse || fetchPromise;
        });
      })
    );
  } else if (requestURL.pathname === "/~manekic/pwa/rezultati.html") {
    event.respondWith(
      caches.open(CACHE_NAME).then(function(cache) {
        return fetch(event.request).then(function(networkResponse) {
          cache.put(event.request, networkResponse.clone());
          return networkResponse;
        }).catch(function() {
          return caches.match(event.request);
        });
      })
    );
  } else if (requestURL.pathname.startsWith("6")) {
      event.respondWith(
        caches.match(event.request).then(function(response) {
          return response || fetch(event.request);
        })
      );
    } else if (
    CACHED_URLS.includes(requestURL.href) ||
    CACHED_URLS.includes(requestURL.pathname)
  ) {
    event.respondWith(
      caches.open(CACHE_NAME).then(function(cache) {
        return cache.match(event.request).then(function(response) {
          return response || fetch(event.request);
        });
      })
    );
  }
});
```

Do sada smo sve skripte koje koristi naša aplikacija spremili u predmemoriju te ih,

neovisno o mreži, možemo otvoriti bez da nam preglednik prikaže poruku o prekidu internetske veze kao na slici 0.1. To je prvi korak ka tome da omogućimo rad naše aplikacije korisnicima koji nemaju pristup internetu. Sljedeća važna stavka je upoznati se s lokalnom pohranom podataka.

Poglavlje 4

Lokalno spremanje podataka

Iako smo tehnikama opisanim u prethodnom poglavlju osigurali da studenti mogu otvoriti stranicu *rezultati.html* kada nemaju pristup internetu, nismo osigurali da se na istoj stranici zaista prikažu odgovarajući rezultati studenata. Razlog tome je taj što se aplikacija za svaku manipulaciju podacima (u ovom slučaju radi se o dohvaćanju rezultata) oslanja na Ajax upite upućene poslužitelju. Da bismo postigli neovisnost od poslužitelja u aplikaciji, potrebno je lokalno spremiti rezultate studenata. Postoji nekoliko lokalnih spremnika podataka, a mi ćemo se upoznati s *IndexedDB*.

4.1 Općenito u IndexedDB

IndexedDB je transakcijska, indeksirana i objektno orijentirana baza podataka u pregledniku. Objasnimo redom značenje svakog od navedenih pojmova u ovoj definiciji.

- *IndexedDB je transakcijska*: sve manipulacije ovom bazom podataka su grupirane u transakcije. To znači da će ili sve akcije uspjeti ili niti jedna akcija neće uspjeti. Primjerice, recimo da imamo bazu podataka koja sprema podatke o stanju računa svojih korisnika. Želimo Ivanu oduzeti 200 kuna s računa te ih uplatiti na Markov račun. To su dvije odvojene akcije koje moramo napraviti u bazi podataka. Dakle, moramo od Ivana oduzeti 200 kuna te Marku dodati 200 kuna na račun. Imamo dvije mogućnosti za neuspjeh.

    1. Prva akcija nije uspjela jer Ivan na računu ima samo 50 kuna, a druga akcija je uspješna. Očito smo banku oštetili za 200 kuna koje smo dodali Marku.
    2. Prva akcija je uspjela i Ivanu smo oduzeli 200 kuna, no Markov račun je zamrznut i ne možemo mu dodati željenu svotu. Sada smo Ivana oštetili za 200 kuna koje više ne postoje.

U indexedDB takvih problema nema jer se akcije grupiraju u transakcije. Time postižemo da su sve akcije ili uspjele ili niti jedna od njih nije uspjela.

- *IndexedDB je objektno orijentirana*: za razliku od relacijski baza podataka (primjerice, MySQL) koje se sastoje od tablica s unaprijed definiranim stupcima, objektno orijentirana baza podataka pohranjuje objekte. Svaka baza može sadržavati više spremišta objekata (engl. object store) i svaki od njih može sadržavati više objekata. Baza podataka za prethodni primjer, s Markom i Ivanom, može, primjerice, sadržavati spremište objekata za klijente banke. Unutar tog spremišta svakog klijenta predstavlja jedan objekt koji se sastoji od imena i prezimena klijenta, stanja njegovog računa i slično.

- *IndexedDB je indeksirana*: poput tradicionalnih relacijski baza podataka, i IndexedDB koristi indekse. Svakom spremištu objekata možemo dodati indeks koji koristimo za dohvaćanje željenih objekata.

  U dosad spomenutom primjeru, za spremište objekata koje sadrži klijente banke, kao indeks možemo koristiti prezime klijenta (pod pretpostavkom da ne postoji više klijenata s istim prezimenom).

- *IndexedDB je baza podataka u pregledniku*: IndexedDB je potpunosti smještena unutar preglednika. Bilo koji podatak pohranjen u toj bazi možemo dohvatiti neovisno o stanju mreže. Sve radnje vezane uz IndexedDB pisane su jezikom JavaScript. Objekti koji se pohranjuju mogu biti JavaScript objekti, brojevi, stringovi, polja, regularni izrazi, null vrijednosti i slično.

4.2 Sintaksa IndexedDB

Za početak ćemo se upoznati s akcijama koje se mogu izvoditi u IndexedDB, a koje ćemo koristiti unutar naše aplikacije.

Otvaranje baze podataka

```
var request = window.indexedDB.open("new-db", 1);
request.onerror = function(event) {
  console.log("DB Error " + event.target.error);
};
request.onsuccess = function(event) {
  db = event.target.result;
  console.log("DB: " + db);
}
```

windows.indexedDB.open vraća zahtjev za otvaranje komunikacije s bazom. Osluškuju se događaji *success* i *error* vezani uz taj zahtjev.

Ukoliko *new-db* ne postoji, čim pokrenemo ovaj kôd, u pregledniku će se stvoriti i otvoriti nova baza podataka tog imena. Ukoliko navedena baza već postoji, ona će se pokretanjem koda samo otvoriti. Aktivira se događaj *success* te ispisuju informacije o bazi.

Kreiranje spremišta objekata

IndexedDB je verzionirana. To znači da, želimo li kreirati novo spremište objekata ili samo modificirati postojeće, moramo započeti komunikaciju s bazom podataka koristeći novu verziju. Dodamo li gore spomenutom kodu sljedeće izmjene, kreirat ćemo spremište objekata pod nazivom *članovi*. Također, zadali smo da je *prezime* ključ svakog objekta u tom spremištu.

```
var request = window.indexedDB.open("new-db", 2);
request.onupgradeneeded = function(event) {
  var db = event.target.result;
  db.createObjectStore("clanovi", {keyPath: "prezime"});
};
```

Kada preglednik uoči da da je broj verzije veći nego kod kreiranja baze podataka, on aktivira događaj *upgrade needed* te na taj način modificira postojeću bazu podataka.

Dodavanje podataka u spremište objekata

Da bismo dodali podatke u spremište objekata, moramo pokrenuti transakciju. To se postiže pozivanjem funkcije *transaction* na objektu koji predstavlja bazu podataka. Naravno, prije toga je potrebno otvoriti željenu bazu. Transakcije u IndexedDB mogu biti *readwrite* ili *readonly* te stoga funkcija *transaction* prima dva parametra. Prvi je ime spremišta u kojeg želimo dodati elemente, a drugi je neobavezan i označava da li je transakcija *readonly* (zadano) ili *readwrite*.

```
var request = window.indexedDB.open("new-db", 2);
request.onsuccess = function(event) {
  var db = event.target.result;
  var podaci = [
    {"prezime": "Horvat", "ime": "Karlo", "id": "0123"},
    {"prezime": "Anic", "ime": "Ana", "id": "4567"}
  ];
```

```
  var transakcija = db.transaction("clanovi", "readwrite");
  transakcija.onerror = function(event) {
    console.log("Error: " + event.target.error);
  };
  var spremiste = transakcija.objectStore("clanovi");
  for(var i = 0; i < podaci.length; i++)
    spremiste.add(podaci[i]);
};
```

Dohvaćanje podataka iz baze podataka

Podatke iz baze možemo dohvatiti na tri načina: korištenjem pokazivača (engl. cursor), korištenjem ključa objekta te korištenjem indeksa. U nastavku slijedi demonstracija sva tri načina.

- **Dohvaćanje objekata pomoću ključa**. Prilikom kreiranja spremišta *članovi*, kreiran je ključ *prezime*. To znači da svaki objekt možemo dohvatiti pomoću tog ključa.

  ```
  var request = window.indexedDB.open("new-db", 2);
  request.onsuccess = function(event) {
    db = event.target.result;
    var transakcija = db.transaction("clanovi", "readonly");
    var spremiste = transakcija.objectStore("clanovi");
    var req = spremiste.get("Horvat");
    req.onsuccess = function(event) {
      var clan = event.target.result;
      console.log("Ime trazenog clana je " + req.ime);
      //ispis: "Ime trazenog clana je Karlo"
    };
  };
  ```

  Pokrenemo li gornji kôd, nakon otvaranja baze podataka, pokreće se transakcija na spremištu *članovi*. Ovog puta funkciji *transaction* kao drugi parametar proslijeđujemo *readonly* jer ne namjeravamo raditi pohranu u spremište objekata. Iz našeg spremišta članovi ćemo pomoću funkcije *get* dohvatiti objekt čiji ključ je prezime *Horvat*. Kako *get* ima asinkrono djelovanje, ono ne vraća odmah rezultat, već vraća objekt koji predstavlja naš zahtjev. Osluškivanjem događaja *success* za ovaj zahtjev, čekamo objekt koji smo zatražili. Član *Karlo Horvat* se nalazi u spremištu pa će se kao ime traženog člana ispisati *Karlo*.

- **Dohvaćanje objekata pomoću pokazivača**. Dohvaćanjem objekata pomoću ključa možemo dohvatiti najviše jedan objekt i moramo mu znati točan ključ. Zaključujemo da ta metoda nije uvijek najbolja. Dohvaćanje više objekata odjednom možemo postići korištenjem pokazivača.

```
var request = window.indexedDB.open("new-db", 2);
request.onsuccess = function(event) {
  db = event.target.result;
  var transakcija = db.transaction("clanovi", "readonly");
  var spremiste = transakcija.objectStore("clanovi");
  var kursor = spremiste.openCursor();
  kursor.onsuccess = function(event) {
    var k = event.target.result;
    if(!k) { return; }
    console.log("Ime trazenog clana je " + k.value.ime);
    //ispis u 1.prolazu: "Ime trazenog clana je Karlo"
    //ispis u 2.prolazu: "Ime trazenog clana je Ana"
    k.continue();
  };
};
```

Kao i u kodovima do sada, prvo moramo otvoriti bazu podataka i pokrenuti transakciju nad željenim spremištem. Nakon toga otvaramo pokazivač, asinkronu funkciju koja pokreće događaj *success* svaki puta kada se pokazivač pomakne (pomoću funkcije *continue*). Unutar funkcije *onsuccess* možemo pristupiti pokazivaču kako bismo dohvatili objekt na kojeg pokazivač trenutno pokazuje. U konzolu ispisujemo vrijednost pokazivača koja nas zanima i pomičemo se na idući objekt pozivajući funkciju *conutine* na pokazivaču.

Kao što je već spomenuto, prilikom svakog pomicanja pokazivača, pokreće se događaj *success*. To znači da će se nakon **svakog** pomicanja pokazivača, pa čak i onda kada pokazivač pokazuje na zadnji zapis ili čak i onda kada je spremište prazno, pokrenuti događaj *success*. Zbog toga je, prije pristupa nekoj vrijednosti, važno uvijek provjeriti pokazuje li zaista pokazivač na nešto.

- **Dohvaćanje objekata pomoću indeksa**. U dohvaćanju objekata pomoću pokazivača smo vidjeli kako iterirati po svim objektima i dohvaćati željene vrijednosti. No, kada želimo dohvatiti samo objekte koji zadovoljavaju neki uvjet (primjerice, sve osobe koje su rođene 1994.), tada moramo koristiti indekse. Napravimo sada novu bazu podataka i novo spremište objekata pod imenom *osobni_podaci* te spre-

mimo u njega nekoliko objekata, ali tako da više njih ima godinu rođenja jednaku 1994.

```
var request = window.indexedDB.open("db", 1);
request.onupgradeneeded = function(event) {
  var db = event.target.result;
  var spremiste = db.createObjectStore("osobni_podaci",
                                       {autoIncrement: true});
  spremiste.createIndex("godina_rodenja", "godina_rodenja",
                        {unique: false});
};
request.onsuccess = function(event) {
  var db = event.target.result;
  var podaci = [
    {"godina_rodenja": "1994", "ime": "Maja"},
    {"godina_rodenja": "1995", "ime": "Hana"},
    {"godina_rodenja": "1994", "ime": "Marko"},
    {"godina_rodenja": "1994", "ime": "Ivan"}
  ];
  var transakcija = db.transaction("[osobni_podaci]", "readwrite");
  transakcija.onerror = function(event) {
    console.log("Error: " + event.target.error);
  };
  var op = transakcija.objectStore("osobni_podaci");
  for(var i = 0; i < podaci.length; i++)
    op.add(podaci[i]);
};
```

U prethodnom kodu možemo uočiti vrlo malu razliku u sintaksi koja je namjerno napravljena kako bi se demonstriralo zapisivanje željenog na dva načina.

Spremište *osobni_podaci* kreirali smo s ključem koji se automatski povećava. Za razliku od spremišta *članovi*, gdje je *prezime* bio jedinstven ključ (što baš i nije najsretnije rješenje jer postoji više ljudi istog prezimena, no napravili smo primjer gdje to izbjegavamo), ovdje se jedinstveni ključ za svaki objekt generira automatski. Točnije, prvi objekt će imati identifikacijski broj 1, drugi 2 i tako dalje.

Funkcija *createIndex* primi ime indeksa kao svoj prvi argument, put do ključa koji bi indeks trebao koristiti kao drugi te kao treći argument primi polje koje je neobavezno. To polje govori da ključevi koje indeks koristi nisu jedinstveni, točnije, da više objekata može imati isti ključ. Ostatak koda nam je poznat.

```
var request = window.indexedDB.open("db", 1);
request.onsuccess = function(event) {
  var db = event.target.result;
  var spremiste = db.transaction("osobni_podaci", "readonly")
                    .objectStore("osobni_podaci");
  var indeks = spremiste.index("godina_rodenja");
  var kursor = indeks.openCursor("1994");
  kursor.onsuccess = function(event) {
    var k = event.target.result;
    if(!k) { return; }
    console.log("Ime osobe rodene 1994.godine je " + k.value.ime);
    //ispis u 1.prolazu: "Ime osobe rodene 1994.godine je Maja"
    //ispis u 2.prolazu: "Ime osobe rodene 1994.godine je Marko"
    //ispis u 3.prolazu: "Ime osobe rodene 1994.godine je Ivan"
    k.continue();
  };
};
```

Umjesto da svaku funkciju pišemo zajedno kao samostalnu, ovdje vidimo mogućnost vezanja funkcija jednu na drugu. Također, prikazan je način na koji se može ostvariti ranije spomenuta funkcionalnost.

Ažuriranje objekata u spremištu objekata

Ako nam je poznat primarni ključ objekta, lako ga možemo ažurirati korištenjem funkcije *put*. Funkcija se poziva na spremištu objekta i proslijeđuje mu ažuriranu vrijednost objekta zajedno s primarnim ključem (kako bi se znalo koji objekt se ažurira). U sljedećem kodu je vidljivo kako se implementira opisani postupak.

```
var request = window.indexedDB.open("db", 1);
request.onsuccess = function(event) {
  var db = event.target.result;
  var spremiste = db.transaction("osobni_podaci", "readonly")
                    .objectStore("osobni_podaci");
  var novo = {"godina_rodenja": "1994", "ime": "Maja Lena"};
  var req = spremiste.put(novo, 1);
  req.onsuccess = function() {
    console.log("Ažurirao objekt za koji vrijedi id = 1");
  }:
};
```

Spomenuta funkcija *put* može imati i samo jedan parametar. Naime, dva parametra su potrebna samo u slučaju kada spremište ima definiran auto-inkrementirajući ključ. Ukoliko spremište nema auto-inkrementirajući ključ, tada koristimo *put(object)* poziv funkcije. Ako ključ objekta *object* postoji u bazi podataka, taj objekt će se samo ažurirati, no, ako ključ ne postoji u bazi podataka, u bazu će se dodati objekt *object* s tim ključem. Stoga, kada smo u situaciji da u IndexedDB moramo dodavati objekte, no postoji mogućnost da su oni već spremljeni u istu, najelegantnije je koristiti funkciju *put* (a ne *add*). U tom slučaju izbjegavamo nepotrebne provjere prisutnosti objekta u bazi.

Brisanje objekata iz spremišta objekata

Kod brisanja imamo dvije mogućnosti: možemo obrisati sve objekte iz spremišta ili možemo obrisati samo neke objekte. U prvom slučaju koristimo se funkcijom *clear*, a u drugom funkcijom *delete*.

```
var request = window.indexedDB.open("db", 1);
request.onsuccess = function(event) {
  var db = event.target.result;
  db.transaction("osobni_podaci", "readwrite")
    .objectStore("osobni_podaci").delete(1);
};
```

Ako nam je poznat ključ objekta koje želimo obrisati, to možemo napraviti na način koji je prikazan u prethodnom kodu. Dakle, obrisali smo objekt koji ima ključ 1.

U svim ostalim slučajevima, primjerice, kada želimo obrisati više objekata odjednom, koristimo se pokazivačem i iteracijom istog po objektima iz spremišta. Naravno, potrebno je zadati neku vrijednost po kojoj se određuje hoće li se objekt obrisati iz baze ili neće. U narednom primjeru brisat ćemo sve osobe koje su rođene 1994.godine.

```
var request = window.indexedDB.open("db", 1);
request.onsuccess = function(event) {
  var db = event.target.result;
  var kursor = db.transaction("osobni_podaci", "readwrite")
                 .objectStore("osobni_podaci").openCursor();
  kursor.onsuccess = function(event) {
    var k = event.target.result;
    if(!k) { return; }
    if(k.value.godina_rodenja === "1994")
      k.delete();
    k.continue();
```

```
  };
};
```

Želimo li obrisati sve objekte iz nekog spremišta, funkcijom *clear* ćemo dobiti povratnu vrijednost s događajima *success* i *error*.

```
var request = window.indexedDB.open("db", 1);
request.onsuccess = function(event) {
  var db = event.target.result;
  var brisanje = db.transaction("osobni_podaci", "readwrite")
                   .objectStore("osobni_podaci").clear();
  brisanje.onsuccess = function(event) {
    console.log("Uspjesno brisanje svih objekata!");
  };
};
```

4.3 Lokalna baza podataka za našu aplikaciju

Do sada smo promotrili sve akcije koji će nam trebati za kreiranje i manipulaciju lokalnom bazom podataka za našu aplikaciju i možemo krenuti s radom. Kreirajmo skriptu *indexedDB.js*. U njoj će se nalaziti kôd pomoću kojeg ćemo kreirati i dodati osnovne podatke u bazu. Skriptu *indexedDB.js* je potrebno uključiti unutar svih skripti koje koriste IndexedDB. Dakle, unutar HTML *tag-a head* u skripte *rezultati.html* i *administrator.html* dodajemo sljedeću liniju koda:

```
<script type="text/javascript" src="indexedDB.js"></script>
```

Kreiranje i otvaranje baze podataka *Studenti* postižemo kodom u nastavku.

```
var db;
var request = window.indexedDB.open("Studenti", 1);
request.onerror = function(event) {
  console.log("DB Error " + event.target.error);
};
request.onsuccess = function(event) {
  db = event.target.result;
  console.log("DB: " + db);
  console.log("Object store names: " + db.objectStoreNames);
};
```

Nakon što smo kreirali bazu pod nazivom *Studenti* (verzija je, naravno, prva), u nju želimo dodati potrebna spremišta objekata. Kako se skripta *indexedDB.js* izvršava svaki puta kada administrator ili student posjete aplikaciju, želimo osigurati da se spremišta objekata kreiraju samo jednom - tijekom njihove prve posjete aplikaciji. Iz tog razloga, potrebno je provjeriti nalaze li se spremišta objekata već u bazi. Ako ne, kreiramo ih. Inače, ne radimo ništa. Objašnjeno je implementirano sljedećim kodom.

```
request.onupgradeneeded = function(event) {
  var db = event.target.result;
  if(!db.objectStoreNames.contains("rezultati")) {
    var objStore1 = db.createObjectStore("rezultati",
                                         {keyPath: "id"});
    objStore1.createIndex("id_studenta", "id_studenta",
                          {unique: false});
  }
  if(!db.objectStoreNames.contains("kolegiji"))
    var objStore2 = db.createObjectStore("kolegiji",
                                         {keyPath: "id_kolegija" });
  if(!db.objectStoreNames.contains("novi_rezultati"))
    var objStore3 = db.createObjectStore("novi_rezultati",
                                         {autoIncrement: true });
  if(!db.objectStoreNames.contains("bodovi")) {
    var objStore4 = db.createObjectStore("bodovi",
                                         {keyPath: "id_studenta" });
    objStore4.createIndex("id_kolegija", "id_kolegija",
                          {unique: false});
  }
};
```

U bazi stvaramo spremište objekata pod nazivom *rezultati* te kao ključ stavljamo *id*. Definiramo indeks *id_studenta* za ovo spremište. Put do ključa koji indeks treba koristiti je, također, *id_studenta* te ključ koji indeks koristi nije jedinstven.

Još jedno spremište objekata koje stvaramo ima naziv *kolegiji*. Kao ključ se koristi *id_kolegija*, a indeks nam nije potreban. Na sljedeći način dodajemo popis svih kolegija u spremište *kolegiji*.

```
request.onsuccess = function(event) {
  var db = event.target.result;
  var podaci = [
    {id_kolegija: 1, naziv: "Matematicka analiza 1"},
```

```
    {id_kolegija: 2, naziv: "Matematicka analiza 2"},
    {id_kolegija: 3, naziv: "Linearna algebra 1"},
    {id_kolegija: 4, naziv: "Linearna algebra 2"}
  ];
  var transakcija = db.transaction(["kolegiji"], "readwrite");
  transakcija.onerror = function(event) {
    console.log("Error: " + event.target.error);
  };
  var kolegijiStore = transakcija.objectStore("kolegiji");
  for(var i = 0; i < podaci.length; i++) {
    kolegijiStore.put(podaci[i]);
  }
};
```

Dakle, jedan objekt u spremištu objekata *kolegiji* je, zapravo, jedan redak tablice *kolegiji* koja se nalazi u bazi podataka *Studenti* na poslužitelju. Spremište se koristi za dohvaćanje popisa svih kolegija u skripti *unosRezultata.html*. Kôd koji to omogućava je sljedeći.

```
var request = window.indexedDB.open("Studenti", 1);
request.onsuccess = function(event) {
  var db = event.target.result;
  var objectStore = db.transaction("kolegiji", "readonly")
                      .objectStore("kolegiji");
  objectStore.openCursor().onsuccess = function(event) {
    var cursor = event.target.result;
    if(cursor) {
        //kreiranje radio buttona s nazivima kolegija ...
        cursor.continue();
    }
    else {
        //kreiranje submit buttona
    }
  };
};
```

Treće spremište koje kreiramo naziva se *bodovi*, ključ je *id_studenta*, a indeks jest *id_kolegija*. Put do ključa kojeg indeks koristi je *id_kolegija* i taj ključ nije jedinstven (jer više objekata može imati isti *id_kolegija*).

Na kraju kreiramo spremište pod nazivom *novi_rezultati* koje ima ključ koji se automatski povećava. Indeks, kao i u spremištu *kolegiji*, nije potreban.

Sljedeći korak koji moramo napraviti jest taj da osiguramo prikaz rezultata studentima čak i onda kada su oni *offline*. U spremište *rezultati* je potrebno pohraniti sve rezultate studenata koji se prijave u aplikaciju. Ako pri nekom sljedećem otvaranju aplikacije student ne bude imao pristup internetu, učitat će mu se rezultati iz spremišta. Naravno, u tom trenutku student možda ima i neke nove rezultate, no, da bi i njih vidio, on aplikaciju mora otvoriti u uvjetima u kojima ima pristup internetu. Sljedećim otvaranjem aplikacije koje uspije kontaktirati poslužitelj (to se događa u slučaju da student ima pristup internetu), podaci u spremištu se ažuriraju. Dakle, koristi se funkcija *put* koja ili sprema novi objekt u spremište (u slučaju da objekt još ne postoji u spremištu objekata) ili ažurira već postojeći objekt novim rezultatima.

Pod pretpostavkom da su rezultati već dohvaćeni s poslužitelja i spremljeni u varijablu *data*, pohrana rezultata u spremište implementirana je kodom:

```
var request = window.indexedDB.open("Studenti", 1);
  request.onsuccess = function(event) {
    var db = event.target.result;
    var objectStore = db.transaction("rezultati", "readwrite")
                        .objectStore("rezultati");
    var index = objectStore.index("id_studenta");
    var count = index.count();
    count.onsuccess = function() {
      for(var i = 0; i <data.rez.length; i++) {
        objectStore.put({id: (id+i), id_studenta: id,
                         kolegij: data.rez[i].ime,
                         kol1: data.rez[i].kol1,
                         kol2: data.rez[i].kol2,
                         zavrsni: data.rez[i].zavrsni,
                         dz1: data.rez[i].dz1, dz2: data.rez[i].dz2,
                         dz3: data.rez[i].dz3, dz4: data.rez[i].dz4});
      }
    }
  };
```

dok se dohvaćanje rezultata iz spremišta obavlja pomoću pokazivača na sljedeći način:

```
var request = window.indexedDB.open("Studenti", 1);
    request.onsuccess = function(event) {
```

```
var db = event.target.result;
var objectStore = db.transaction("rezultati", "readonly")
                    .objectStore("rezultati");
var index = objectStore.index("id_studenta");
var cursor = index.openCursor(id);
cursor.onsuccess = function(event) {
  var cursor = event.target.result;
  if (cursor) {
    //ispis rezultata u tablicu
    cursor.continue();
  }
```

Kako manipulacije podacima koje trebamo prikazati studentima ne koriste niti jednu novu tehnologiju, njih izostavljamo u prikazanim kodovima.

Naredni slučaj u kojem trebamo lokalno pohraniti podatke je unos novih bodova od strane administratora. Zapisivanje bodova u bazu na poslužitelju će se riješiti pomoću pozadinske sinkronizacije, no o tome nešto kasnije. Za sada ćemo se samo baviti pohranjivanjem potrebnih podataka u lokalnu bazu na klijentu.

Administratoru želimo omogućiti unos bodova studentima čak i onda kada on nema pristup internetu, no ovdje ima nekoliko komplikacija. Želimo li administratoru dopustiti da on unosi bodove iz svih kolegija, svim studentima, morali bismo lokalno pohraniti čitave baze koje se nalaze na poslužitelju, a to nije efikasno rješenje jer bismo, u tom slučaju, lokalno morali spremati velike količine podataka. Kako bismo pojednostavili ovu situaciju, odlučujemo se na sljedeće: kada je *offline*, administrator može upisati bodove studentima samo za onaj kolegij za koji ih je upisivao zadnji puta kada je imao pristup internetu.

Promotrimo situaciju kada administrator ima pristup internetu. On mora odabrati kolegij za koji želi unesti bodove studentima. Nakon odabira, u Session Storage se spremaju dvije varijable - identifikacijski broj i naziv odabranog kolegija. U skripti *upisRez.html* se ti podaci dohvaćaju te se nakon komunikacije s poslužiteljem prikazuje popis svih studenata koji su upisali odabrani kolegij, zajedno s bodovima koje su ostvarili iz pojedinih elemenata ocjenjivanja.

U ovom trenutku imamo pristup podacima koji su nam potrebni: naziv kolegija (točnije, identifikacijski broj kolegija) te popis studenata koji su upisali odabrani kolegij, popraćen njihovim dosadašnjim rezultatima. Sada treba napraviti lokalnu pohranu podataka.

Odgovor poslužitelja (dostupan u varijabli *data*) lokalno spremamo u spremište objekata *bodovi* na sljedeći način.

```
var request = window.indexedDB.open("Studenti", 1);
      request.onsuccess = function(event) {
```

```
        var db = event.target.result;
        var objectStore = db.transaction(["bodovi"], "readwrite")
                            .objectStore("bodovi");
        objectStore.clear();
        for(var i = 0; i < data.studenti.length; i++) {
          for(var j = 0; j < data.rez.length; j++) {
            if(data.studenti[i].id === data.rez[j].id) {
              objectStore.add({id_studenta: data.studenti[i].id,
                               naziv_kolegija: naziv_kolegija_Rez,
                               id_kolegija: id_kolegija_Rez,
                               ime: data.studenti[i].ime,
                               prezime: data.studenti[i].prezime,
                               kol1: data.rez[j].kol1,
                               kol2: data.rez[j].kol2,
                               zavrsni: data.rez[j].zavrsni,
                               dz1: data.rez[j].dz1, dz2: data.rez[j].dz2,
                               dz3: data.rez[j].dz3, dz4: data.rez[j].dz4});
            }
          }
        }
      };
```

Dakle, sada se u spremištu nalaze objekti koji predstavljaju studente koji su upisali odabrani kolegij. Prije samog spremanja podataka, obrišemo sve objekte koji se već nalaze u spremištu objekata. To su podaci koje smo prošli put spremili u *bodovi*.

Ako administrator posjeti aplikaciju kada je *offline* te želi unesti bodove studentima, on prvo na svojoj početnoj stranici odabire opciju *Unos rezultata* (jedinu koja mu se nudi). Nakon toga se otvara stranica *upisRez.html* na kojoj se prikaže naziv kolegija za koji je zadnje unosio bodove te popis studenata (s bodovima) koji su upisali taj kolegij.

Svaki administratorov unos novih bodova studentima, bilo da je taj unos napravljen kada je administrator *online* ili *offline*, sprema se u spremište objekata *novi_rezultati*. To znači da su ti bodovi spremljeni samo lokalno, a ne u bazi *Studenti* na poslužitelju.

Pohranu podataka u bazu *Studenti* ćemo omogućiti pomoću pozadinske sinkronizacije u sljedećem poglavlju.

4.4 Indikatori promjene stanja mreže

Ranije je na slici 0.2 prikazano da se sadržaj koji aplikacija pruža korisnicima koji su *online* ne razlikuje od sadržaja koji se pruža korisnicima koji su *offline*. Želimo implemen-

tirati indikatore stanja mreže koji će nam omogućiti da lakše uočimo kako nema razlike u sadržaju koji se prikazuje.

Uz ime i prezime studenta ili oznake za administratora (ovisno o tome tko se prijavio u aplikaciju), želimo postaviti indikatore koji nam govore postoji li pristup internetskoj vezi ili ne, točnije, želimo dodati riječi *online, offline* (ovisno o stanju mreže).

U obzir moramo uzeti sljedeću situaciju. Korisnici mogu aplikaciju otvoriti kada su u stanju *online/offline*, no ono se tijekom rada može promijeniti u stanje *offline/online*. Dakle, trebamo imati detekciju stanja mreže prilikom otvaranja aplikacije, no moramo reagirati i na događaj promjene stanja mreže.

Kostur koda pomoću kojega se može implementirati upravo opisana funkcionalnost slijedi u nastavku.

```
if(navigator.onLine) {
  $(/*id elementa gdje se upisuje*/).html("(online)")
                                    .css("color", "#5cb85c");
}
else {
  $(/*id elementa gdje se upisuje*/).html("(offline)")
                                    .css("color", "red");
  $("span").css("color", "red");
  $("h2").css("color", "red");
}

$(window).on("online", function() {
  $(/*id elementa gdje se upisuje*/).html("(online)")
                                    .css("color", "#5cb85c");
  $("span").css("color", "#5cb85c");
  $("h2").css("color", "#5cb85c");
});

$(window).on("offline", function() {
  $(/*id elementa gdje se upisuje*/).html("(offline)")
                                    .css("color", "red");
  $("span").css("color", "red");
  $("h2").css("color", "red");
});
```

Blokovi *if* i *else* služe za otkrivanje da li se korisnik uoči prijave nalazio *online* ili *offline*. Međutim, ukoliko se promijeni korisnikov status veze, sve dok se ne osvježi stranica, to neće biti vidljivo. Upravo zbog toga postoje reakcije na događaje *online* i *offline*. Na

taj način je moguće u svakom trenutku prepoznati promjenu stanja veze i, sukladno s time, promijeniti indikatore.

U klijentske skripte, po potrebi, možemo dodati neku od modifikacija prikazanog koda. Na primjer, u skripti *rezultati.html* u dani kôd želimo dodati funkcije koje prikazuju obavijesti nakon otvaranja aplikacije, funkcije koje ispisuju tablicu s rezultatima studenta (ovisno o stanju mreže, radi se ili o Ajax pozivu, ili o čitanju podataka iz lokalne baze podataka) te funkciju koju, uslijed promjene stanja mreže iz *offline* u *online*, osvježava stranicu kako bi se osigurao prikaz najnovijih rezultata.

Krenuvši od početne aplikacije, do sada smo napravili nekoliko vrlo važnih izmjena. Studenti koji se jednom prijave u aplikaciju te je napuste bez odjave, u mogućnosti su vidjeti svoje rezultate na ispitima čak i onda kada nemaju osiguran pristup internetu. Također, administrator koji napusti aplikaciju bez odjave može vidjeti početnu stranicu koja mu prikazuje popis radnji koje može obavljati te popis svih studenata koji su upisali kolegij za koji je zadnje unosio bodove u bazu podataka (prilikom odabira *Upis rezultata*). U nastavku ćemo osigurati da administrator može unositi nove rezultate studentima neovisno o mreži. Ostale radnje, dakle unos novih studenata, upisivanje novih kolegija te brisanje upisanih kolegija, administratoru neće biti omogućene ukoliko on nema pristup internetu. Odabir bilo koje od navedenih radnji, bilo da je administrator otvorio aplikaciju u stanju *offline* ili, nakon nekog vremena, iz stanja *online* prešao u stanje *offline*, uzrokovati će prikaz odgovarajuće poruke (o nemogućnosti izvođenja željene radnje) ukoliko internetski pristup nije omogućen.

Poglavlje 5

Pozadinska sinkronizacija

Svi smo se barem jednom našli u situaciji kada smo uspred ispunjavanja nekog upitnika ili obrasca ostali bez internetske veze. Sve podatke koje smo upisivali moramo ponovno upisati jer klik na potvrdnu opciju (engl. submit button) javlja poruku o prekidu internetske veze i time briše sve uneseno.

Rješenje takvih i sličnih situacija nam omogućava pozadinska sinkronizacija. Ona osigurava da se svaka radnja koju korisnik napravi dovrši - bez obzira na stanje mreže.

5.1 Općenito o pozadinskoj sinkronizaciji

Pozadinska sinkronizacija je mehanizam koji s klijentske strane aplikacije pokušava kontaktirati poslužitelja pomoću uslužne skripte. Ako korisnik nema pristup internetu, poslužitelj neće moći biti kontaktiran, no uslužna skripta, u tom slučaju, automatski periodički pokušava kontaktirati poslužitelja i poslati mu odgovarajuće podatke. Na primjer, ako se radi o ispunjavanju nekog obrasca, uslužna skripta mora poslužitelju poslati sve podatke koje je korisnik unesao u obrazac.

Naravno, uslužna skripta negdje mora zapamtiti podatke koje treba poslati poslužitelju. Kako korisnik nema pristup internetu, podaci se spremaju lokalno u bazu IndexedDB. Jednom kada poslužitelj dobije podatke, oni se mogu obrisati iz lokalne baze.

Dakle, omogućavanjem pozadinske sinkronizacije korisnici više ne moraju brinuti hoće li njihove radnje ”doći” do poslužitelja jer je odgovor na to pitanje u svakom slučaju potvrdan - čak i onda kada korisnik zatvori svoj preglednik, uslužna skripta periodički pokušava kontaktirati poslužitelja.

5.2 Omogućavanje pozadinske sinkronizacije

U skripti u kojoj imamo situaciju gdje želimo koristiti pozadinsku sinkronizaciju (kod nas je to, na primjer, skripta *upisRez.html*), moramo registrirati događaj *sync* na sljedeći način:

```
navigator.serviceWorker.ready.then(function(reg) {
    return reg.sync.register('send-msgs');
});
```

Navedeni kôd koristi objekt registracije aktivne uslužne skripte kako bi registrirao događaj *sync* naziva *send-msgs*. U slučaju kompleksnih aplikacija, pozadinska sinkronizacija se može brinuti o više različitih podataka s kojima treba postupati na različite načine. Kako bismo mogli razlikovati događaje *sync*, pridružujemo im različita imena.

Da bismo omogućili pozadinsku sinkronizaciju, također je potrebno uslužnu skriptu pretplatiti na događaj *sync*. Navedeno možemo napraviti dodajući sljedeći kôd u skriptu koja implementira uslužnu skriptu.

```
self.addEventListener('sync', function(event) {
  console.log("Event tag je "+event.tag);
  if (event.tag === 'send-msgs') {
    event.waitUntil(send());
  }
});
```

Uslužna skripta osluškuje na događaj *sync*. Ukoliko je riječ o događaju *sync* naziva *send-msgs*, uslužna skripta pokušava izvršiti funkciju *send* (čija implementacija ovdje nedostaje iz razloga što nije važna). Korištenjem funkcije *waitUntil* osiguravamo da se događaj *sync* ne završi sve dok mu mi to ne kažemo. Dobivamo na vremenu da izvedemo neke akcije te uspješno razriješimo događaj ako one uspiju. Ako akcije ne uspiju, odnosno ako rezultiraju odbijenim *promise-om* (engl. rejected promise), tada će uslužna skripta kasnije ponovno pokušati razriješiti ovaj događaj. Pokušaji će se ponavljati sve dok ne uspiju.

5.3 SyncManager

Svaka interakcija s događajima *sync* odvija se pomoću SyncManager-a. To je sučelje uslužne skripte koje nam dopušta da registriramo različite događaje *sync*. SyncManageru-u možemo pristupiti pomoću objekta registracije aktivne uslužne skripte. Taj objekt možemo

dohvatiti pomoću uslužne skripte ili neke druge skripte naše aplikacije. No, način na koji ga dohvaćamo se razlikuje za ova dva spomenuta slučaja.

Ukoliko registracijskom objektu želimo pristupiti unutar uslužne skripte, sljedeći kôd će nam poslužiti:

```
self.registration
```

Ako pak, s druge strane, registracijskom objektu želimo pristupiti iz neke druge skripte koju kontrolira trenutno aktivna uslužna skripta, to možemo na sljedeći način:

```
navigator.serviceWorker.ready.then(function(registration) {});
```

Nakon što smo dohvatili registracijski objekt, interakcija sa SyncManager-om je identična pristupamo li mu iz uslužne skripte ili neke druge skripte.

SyncManager sadrži listu svih registriranih događaja *sync*. Nazivi, odnosno, oznake događaja (engl. event tags) su jedinstvene. Ako registriramo novi događaj *sync* s oznakom koja već postoji, SyncManager će to ignorirati i neće ponoviti unos te oznake na svoju listu. Upravo to nam omogućava da grupiramo slične događaje.

Primjerice, ako imamo aplikaciju za slanje poruka, potpuno je logično da ne želimo poslati obavijest našem korisniku za svaku poruku koju mu pošalje korisnik A, već ćemo sve poruke primljene od korisnika A grupirati u jednu obavijest.

SyncManager poruke sprema u redove u skladu s njihovim oznakama događaja. Kako smo već naveli, poruke se spremaju lokalno te je vrlo praktično za svaku oznaku događaja kreirati jedno spremište objekata. U trenutku kada uslužna skripta uspije poslati poruku poslužitelju, poruka se briše iz reda, odnosno iz spremišta objekata u koji je bila pohranjena.

5.4 Fetch API

Komunikacija s poslužiteljom se dosad, u većini slučajeva, odvijala pomoću Ajax poziva. Kod pozadinske sinkronizacije slanje podataka poslužitelju obavlja uslužna skripta. To znači da se implementacija istog nalazi unutar uslužne skripte. Sada je vrijeme da se upoznamo s *Fetch API-em*.

Fetch je moderan način za izvođenje Ajax poziva pri radu u jeziku JavaScript. Umjesto pisanja nezgrapnog Ajax koda ili korištenja biblioteka kao što su jQuery i Angular, novi JavaScript standard nudi kompaktniju, modernu i fleksibilnu sintaksu.

```
fetch('server.php', {
    method: 'POST',
    headers: {
```

```
        'Accept': 'application/json',
        'Content-type': 'application/json',
    },
    body:JSON.stringify({a, b, c})
})
.then((res) => res.json())
.then((data) => console.log(data))
.catch((error) => console.log(error))
```

Želimo li poslati podatke poslužitelju, moramo znati *url* skripte kojoj šaljemo podatke, u ovom slučaju to je skripta *server.php*. Potreban je parametar *method* koji govori da se radi o slanju podataka pomoću HTTP metode POST. *body* sadrži podatke koje želimo poslati poslužitelju. Naravno, koristimo funkciju *JSON.stringify* kako bismo objekte pretvorili u string. Parametar *headers* je opcionalan te određuje vrstu podataka koje šaljemo kao i vrstu onih podataka koje primamo. Navedeni kôd vraća *promise* koji se mora razriješiti. Ukoliko se on uspješno razriješi, u konzolu se ispišu podaci koje je poslužitelj poslao kao odgovor na klijentov zahtjev. Ako se pak *promise* neuspješno razriješi, u konzolu se ispiše odgovarajuća poruka koja opisuje razlog neuspjeha.

5.5 Omogućavanje pozadinske sinkronizacije unutar naše aplikacije

Došli smo do dijela kada ćemo omogućiti administratoru da unosi nove rezultate studentima neovisno o njegovom pristupu mreži. Do sada je klijentska strana aplikacije poslužitelju podatke o broju bodova slala pomoću Ajax poziva na skripti *upisRez.html*. Imajući to u vidu, s promjenama prvo krećemo upravo na toj skripti.

Ajax poziv se odvijao unutar funkcije *spremiRezultat* koja se poziva za svaku promjenu *input* polja. Iz razloga što se podaci koje uslužna skripta treba poslati poslužitelju spremaju lokalno, moramo promijeniti logiku tog dijela naše aplikacije.

Podatke o novim rezultatima koje administrator unese u aplikaciju, prvo spremamo u IndexedDB. U odjeljku 4.3, spomenuli smo spremište objekata *novi_rezultati*. Upravo to spremište ćemo koristiti za pohranu novih rezultata.

Zamijenit ćemo kôd navedenog Ajax poziva sljedećim kodom:

```
var request = window.indexedDB.open("Studenti", 1);
request.onsuccess = function(event) {
  var db = event.target.result;
  var objectStore = db.transaction(["novi_rezultati"], "readwrite")
                      .objectStore("novi_rezultati");
```

```
  objectStore.add({id_studenta: id_studenta,
                   naziv_kolegija: naziv_kolegija_Rez,
                   id_kolegija: id_kolegija_Rez, elt: elt,
                   bodovi: bodovi});
 }
 navigator.serviceWorker.ready
  .then(function(registration) {
    registration.sync.register('upis-rezultata');
  }).then(function() {
    console.log("Sync registered");
  }).catch(function(error) {
    console.log("It broke").
    console.log(error.message);
 });
```

Registrirali smo događaj *sync* te mu dodijelili oznaku *upis-rezultata*. U slučaju uspješne registracije, u konzolu se ispiše popratna poruka. Ako pak registracija krene po zlu, u konzolu će se također ispisati odgovarajuća poruka.

Potrebno je kreirati novu poslužiteljsku skriptu koju će uslužna skripta kontaktirati kako bi osigurala pohranu podataka o broju bodova iz lokalne baze u bazu *Studenti* na poslužitelju. Skriptu ćemo nazvati *server.php*. Ona se, uglavnom, sastoji od koda kao i skripta koja se pozivala unutar Ajax poziva, no ima neke izmjene. *server.php* izgleda ovako:

```
<?php
require_once 'db.class.php';

  $data = json_decode(file_get_contents('php://input'), true);
  $id_studenta = $data['id_studenta'];
  $id_kolegija = $data['id_kolegija'];
  $bodovi = $data['bodovi'];
  $elt = $data['elt'];

  //spajanje na bazu, tablica rezultati
  try {
    //imamo 7 mogućih slučajeva - ovisno o
    //elementu ocjenjivanja o kojem se radi
    if($elt == "1kolokvij") {
      $db = DB::getConnection();
      $st = $db->prepare("UPDATE rezultati SET 1kolokvij =:bodovi
```

```
                        WHERE student_id =:student_id
                        AND kolegij_id =:kolegij_id");
      $st->execute(array('bodovi' => $bodovi, 'student_id' => $id_studenta,
                        'kolegij_id' => $id_kolegija));
    $message['podrucje'] = "1. kolokvija";
    }
    //ostalih 6 slučajeva analogno ...
   }
    catch( PDOException $e ) { exit( 'PDO error ' . $e->getMessage() ); }
 ?>
```

Dalje trebamo pretplatiti uslužnu skriptu na događaj *sync* kako bi ga ona počela osluškivati. U skriptu *sw.js* dodajemo sljedeći kod:

```
self.addEventListener('sync', function(event) {
  if (event.tag === 'upis-rezultata') {
    event.waitUntil(pohranaUBazu());
  }
});

function pohranaUBazu() {
  var request = indexedDB.open("Studenti", 1);
  request.onsuccess = function(event) {
    var db = event.target.result;
    var objectStore = db.transaction("novi_rezultati", "readwrite")
                        .objectStore("novi_rezultati");
    var cursor = objectStore.openCursor();
    cursor.onsuccess = function(event) {
      var cursor = event.target.result;
      if (!cursor) { return; }
      var id_studenta = cursor.value.id_studenta;
      var id_kolegija = cursor.value.id_kolegija;
      var bodovi = cursor.value.bodovi;
      var elt = cursor.value.elt;
      fetch('server.php', {
        method: 'POST',
        headers: {
            'Accept': 'application/json',
            'Content-type': 'application/json',
        },
```

```
        body: JSON.stringify({
          id_studenta: id_studenta,
          id_kolegija:id_kolegija,
          bodovi: bodovi,
          elt: elt})
      })
      .catch(function(err) {
        console.log('It broke' + err.messag);
      });
      if(cursor.value.id_studenta === id_studenta)
        cursor.delete();
      cursor.continue();
    };
  };
}
```

Unutar uslužne skripte potrebno je, pomoću funkcije *fetch* poslati podatke o broju bodova poslužitelju. Naravno, prije samog slanja, potrebno je iste podatke dohvatiti iz lokalne baze podataka. Podatke dohvaćamo pokazivačem. Nakon svakog dohvata, objekt funkcijom *JSON.stringify* pretvaramo u string, šaljemo ga poslužitelju i brišemo ga iz lokalne baze. Pokazivač se nakon svakog dohvata pomiče.

U ovom trenutku imamo jako naprednu aplikaciju. Korisnicima je omogućen pristup aplikaciji neovisno o stanju mreže, a osigurali smo i to da se ne moraju brinuti o tome hoće li se njihove radnje zaista dostaviti serveru. Jedina značajka koja je ostala za implementaciju u svrhu pretvaranja naše aplikacije u PWA jest tzv. push obavijest. Push obavijest želimo poslati svakom korisniku nakon što administrator za njega unese nove rezultate u bazu podataka na poslužitelju.

Poglavlje 6

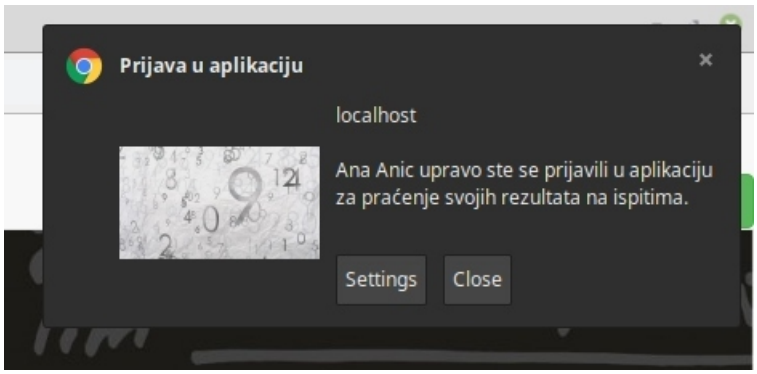
Push obavijesti

6.1 Općenito o push obavijestima

Došlo je vrijeme da se upoznamo sa značajkom PWA koja vlasniku aplikacije pomaže zadržati korisnike. Kombiniranjem *Push API-a* s *Notification API-em* omogućavamo slanje push obavijesti korisnicima čak i onda kada su oni napustili aplikaciju.

Notification API

Notification API omogućava web-aplikaciji da prikazuje obavijesti korisnicima. Obavijesti se prikazuju izvan preglednika, najčešće u malom skočnom prozoru kao na slici 6.1, a za njihovo prikazivanje je potrebno dopuštenje korisnika.



Slika 6.1: Obavijest korisniku

Dopuštenje se može dobiti pokretanjem sljedećeg koda koji se smješta u skriptu u kojoj želimo od korisnika zatražiti isto (kod nas je to, na primjer, skripta *rezultati.htm*).

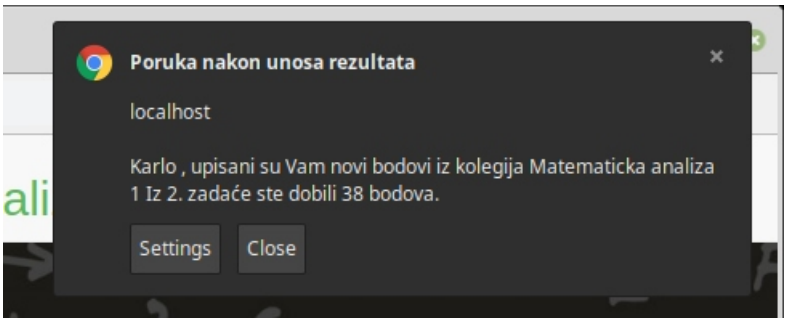
```
Notification.requestPermission().then(function(permission) {
    if(permission === "granted") {
        navigator.serviceWorker.ready.then(function(registration) {
            registration.showNotification("Prva obavijest");
        )};
    }
    else if(permission === "denied") {
        console.log("Odbili ste obavijesti.");
    }
});
```

Korisnik može prihvatiti ili odbiti obavijesti. Ukoliko ih prihvati, varijabla *permission* poprimit će vrijednost *granted* te će se korisniku prikazati obavijest pod nazivom *Prva obavijest*. U suprotnom, varijabla će imati vrijednost *denied* i u konzolu će se ispisati informacija o neprihvaćanju obavijesti.

Push API

Push API omogućava korisnicima da se pretplate na push obavijesti iz naše aplikacije te dopušta poslužitelju da šalje preglednicima pretplaćenih korisnika poruke u bilo koje vrijeme. Porukama upravlja uslužna skripta koja osluškuje na događaje povezane s njima te djeluje sukladno sa zahtjevima koje definiraju događaji. Ti zahtjevi su, najčešće, slanje obavijesti korisnicima.

Push obavijesti se, poput obavijesti, prikazuju u skočnom prozoru izvan preglednika. Na slici 6.2 je vidljivo da se njihov izgled ne razlikuje mnogo.

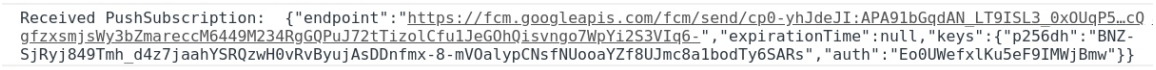


Slika 6.2: Push obavijest korisniku

U push obavijestima sudjeluju tri strane, poslužitelj za našu aplikaciju, preglednik u kojem je aplikacija otvorena te centralni poslužitelj poruka. Postoji nekoliko ključnih koraka koja se moraju dogoditi da bi se korisnik uspješno pretplatio na push obavijesti.

Prvi korak u pretplaćivanju korisnika na push obavijesti je pribavljanje dopuštenja za prikazivanje obavijesti te kontaktiranje centralnog poslužitelja poruka od kojeg tražimo kreiranje nove pretplate za korisnika. Pisanje koda za centralni poslužitelj poruka nije potrebno jer je je to usluga koju tipično pružaju proizvođači preglednika poput Google-a i Mozilla-e. Centralni poslužitelj poruka pohranjuje detalje pretplate te ih vraća našoj aplikaciji.

Izgled detalja o pretplati možemo vidjeti na slici 6.3. Jednom kada smo zaprimili te detalje, potrebno ih je pohraniti u bazu podataka na poslužitelju kako bi im poslužitelj mogao pristupiti u svakom trenutku.



Slika 6.3: Detalji pretplate na push obavijesti

Kada odlučimo poslati poruku korisniku, poslužiteljska skripta dohvaća podatke o pretplati tog korisnika iz baze podataka. Podaci se koriste za slanje poruke centralnom poslužitelju poruka koji, po primitku istih, poruku prosljeđuje korisniku. Na kraju, uslužna skripta koja je registrirana u korisnikovom pregledniku, te koja osluškuje na događaje *push*, prima poruku, čita njezin sadržaj i prikazuje je korisniku.

6.2 VAPID ključevi

Prilikom pretplaćivanja korisnika na push obavijesti, centralni poslužitelj poruka vraća detalje pretplate koji sadrže sve potrebne informacije za slanje bezbroj poruka dotičnom korisniku. Možemo se naći u situaciji da neka skripta nevezana za našu aplikaciju ”presretne” te detalje o pretplati korisnika te ih zlorabi šaljući korisniku neželjene obavijesti. Kako bismo izbjegli spomenuti problem, centralni poslužitelj poruka prihvaća samo one poruke koje sadrže privatni ključ koji je pohranjen na našem poslužitelju. Da bi se potvrdilo da poruke sadrže točan privatni ključ, za svaki privatni postoji odgovarajući javni ključ. Javni ključ se nalazi unutar naše skripte te se šalje centralnom poslužitelju poruka zajedno sa zahtjevom za novu pretplatu. Na taj način centralni poslužitelj poruka pohranjuje javni ključ zajedno s detaljima pretplate na push obavijesti.

Dakle, prilikom stvaranja aplikacije, potrebno je generirati privatni i javni ključ koje će aplikacija koristiti u svrhu slanja push obavijesti. Sami postupak generiranja ključeva će biti opisan u odjeljku 6.4. VAPID ključevi je oznaka za privatni i odgovarajući javni ključ. Isti VAPID ključevi se koriste za sve korisnike jer su vezani uz poslužiteljski dio aplikacije koji generira push obavijesti.

6.3 Web Push PHP

Podaci koji se šalju unutar push obavijesti moraju biti šifrirani. Također, da bi preglednik mogao ispravno dešifrirati poruku, podacima koji se šalju potrebno je dodati određena zaglavlja. Iz razloga što je jako komplicirano da sami šifriramo i dešifriramo poruke, pri slanju push obavijesti koristimo biblioteke koje taj posao rade umjesto nas.

Obzirom da poslužiteljske skripte pišemo u jeziku PHP, biblioteka mora odgovarati tome. Biblioteka koju ćemo mi koristiti je web-push-php [6]. Potrebno ju je preuzeti na vlastito računalo na način koji je opisan u uputama (pomoću composer-a [2]). U opisu biblioteke detaljno je objašnjeno što joj je sve potrebno za ispravan rad. U nastavku opisujemo kako pomoću ove biblioteke implementirati obavijesti unutar naše aplikacije.

6.4 Obavijesti i push obavijesti u našoj aplikaciji

Obavijesti

Studentima želimo prikazati obavijest svaki puta kada otvore našu aplikaciju. Ako su aplikaciju otvorili onda kada su *offline*, u obavijesti im želimo poručiti da im se možda ne prikazuju najnoviji rezultati.

Implementaciju zapisujemo u skriptu *rezultati.html*. Kao što je već spomenuto u odjeljku 4.3, u skripti imamo blokove koji detektiraju *online*, odnosno, *offline* prijavu. Unutar ta dva bloka pozvat ćemo funkcije *notifikacijeOnline* i *notifikacijeOffline*, redom, kako bismo korisnicima prikazali odgovarajuće obavijesti.

Na početku skripte dohvaćamo ime i prezime studenta te ih pamtimo u varijablama *ime* i *prezime*. Funkcija *notifikacijeOnline* implementirana je sljedećim kodom:

```
function notifikacijeOnline() {
    if("Notification" in window && "serviceWorker" in navigator) {
      Notification.requestPermission().then(function(permission) {
        if(permission === "granted") {
          console.log("granted");
          navigator.serviceWorker.ready.then(function(registration) {
            registration.showNotification("Prijava u aplikaciju", {
              body: ime +" "+prezime+" "+"upravo ste se prijavili u
                    aplikaciju za praćenje svojih rezultata na ispitima.",
              icon: "4.jpg",
              tag: "novi-login"
            });
          });
```

```
        }
      });
    }
}
```

Kôd funkcije *notifikacijeOffline* je sličan kao gore navedeni. Jedina razlika se nalazi u *body-u* obavijesti u kojemu se nalazi sljedeći tekst:

```
body: ime +" "+prezime+" "+"upravo ste se prijavili u aplikaciju
      za praćenje svojih rezultata na ispitima. Upozorenje: možda
      Vam se ne prikazuju najnoviji rezultati. Prijavite se u aplikaciju
      kada ćete imati pristup internetu za dohvat najnovijih rezultata!",
```

Kao što je vidljivo iz koda funkcija, prvo se provjerava jesu li uslužna skripta i obavijesti podržane u pregledniku. Ako jesu, pribavlja se dopuštenje za obavijesti te se šalje odgovarajuća obavijesti.

Obavijestima smo definirali *body*, *icon* te *tag*, a ti argumenti redom označavaju sadržaj obavijesti, URL slike koju želimo korisnicima prikazati uz obavijest te jedinstveni identifikator ove obavijesti. Kao dodatni atributi još se mogu navesti *actions* (obavijesti možemo dodati do dva izbora na koja korisnici mogu kliknuti; primjerice potvrdnu opciju), *sound* (URL audio datoteke za koju želimo da se otvori kada se obavijest kreira), *data* (pridruživanje podataka obavijestima) i slično.

Push obavijesti

Krenut ćemo s preuzimanjem biblioteke web-push-php. U Linux terminalu potrebno se smjestiti u korijenski direktorij našeg projekta te upisati naredbu

```
composer require minishlink/web-push
```

koja će stvoriti datoteke i direktorije potrebne za rad biblioteke. Nakon toga je potrebno generirati VAPID ključeve. U tu svrhu u korijenskom direktoriju projekta dodajemo direktorij naziva *keys*. Kako bismo generirali ključeve, u Linux terminalu se smjestimo unutar upravo kreiranog direktorija te upisujemo naredbe:

```
$ openssl ecparam -genkey -name prime256v1 -out private_key.pem
$ openssl ec -in private_key.pem -pubout -outform DER|tail -c 65|base64
  |tr -d '=' |tr '/+' '_-' >> public_key.txt
$ openssl ec -in private_key.pem -outform DER|tail -c +8|head -c 32|base64
  |tr -d '=' |tr '/+' '_-' >> private_key.txt
```

koje generiraju odgovarajuće datoteke te, unutar njih, upisuju odgovarajuće ključeve.

Unutar *$(document).ready()* u skripti *rezultati.html* dodajemo sljedeći kod:

```
if(!("Notification" in window)) {
    console.error("This browser does not support desktop notification");
}

else if (Notification.permission === "granted") {
    console.log("Dozvola za primanje obavijesti je prihvaćena!");
}

else if (Notification.permission !== 'denied') {
    Notification.requestPermission(function (permission) {
      if (permission === "granted") {
            console.log("Dozvola za primanje obavijesti je prihvaćena!");
            push_pretplata();
      }
    });
}
```

Započinje se provjerom omogućava li preglednik prikazivanje obavijesti, ako da, provjerava se je li pribavljeno dopuštenje za prikazivanje obavijesti. U slučaju da dozvola za prikazivanje obavijesti nije pribavljena, potrebno ju je pokušati pribaviti te, nakon toga, pozivanjem funkcije *push_pretplata*, pretplatiti korisnika na push obavijesti. Ukoliko je dozvola bila pribavljena ranije, korisnik je već pretplaćen na push obavijesti te se tada navedena funkcija ne poziva.

```
function push_pretplata() {
    return dohvacanje_dozvole()
      .then(() => navigator.serviceWorker.ready)
      .then(serviceWorkerRegistration =>
        serviceWorkerRegistration.pushManager.subscribe({
          userVisibleOnly: true,
          applicationServerKey: urlBase64ToUint8Array(/*javni ključ*/),
        })
      )
      .then(subscription => {
          return push_slanje_serveru(subscription, 'POST');
      });
}
```

```
function dohvacanje_dozvole() {
  return new Promise((resolve, reject) => {
    if(Notification.permission === 'denied') {
      return reject(new Error('Push obavijesti blokirane.'));
    }
    if(Notification.permission === 'granted') {
      return resolve();
    }
    if (Notification.permission === 'default') {
      return Notification.requestPermission().then(result => {
        if(result !== 'granted') {
          reject(new Error('Nije potvrdjeno!'));
        }
        resolve();
      });
    }
  });
}
```

Funkcija *push_pretplata* prvo dohvaća status o dozvoli za primanje obavijesti te pretplaćuje korisnika na push obavijesti. Da bi se korisnik pretplatio na push obavijesti, potrebno je obavezni atribut *userVisibleOnly* postaviti na *true*. To znači da sve push obavijesti moraju biti vidljive korisniku. Centralni poslužitelj poruka nam vraća detalje o pretplati korisnika koje moramo spremiti u bazu podataka. Funkcija *urlBase64ToUint8Array* radi pretvorbu javnog ključa u format koji centralni poslužitelj poruka raspoznaje. Implementacija funkcije dana je sljedećim kodom.

```
function urlBase64ToUint8Array(base64String) {
    const padding = '='.repeat((4 - (base64String.length % 4)) % 4);
    const base64 = (base64String + padding).replace(/\-/g, '+')
                                           .replace(/_/g, '/');

    const rawData = window.atob(base64);
    const outputArray = new Uint8Array(rawData.length);

    for (let i = 0; i < rawData.length; ++i) {
        outputArray[i] = rawData.charCodeAt(i);
    }
```

```
    return outputArray;
}
```

Funkcijom *push_slanje_serveru*, koja je implementirana narednim kodom, pohranjuju se informacije o pretplati korisnika na push obavijesti u bazu podataka na našem poslužitelju.

```
function push_slanje_serveru(subscription, method) {
    const key = subscription.getKey('p256dh');
    const token = subscription.getKey('auth');
    const contentEncoding = (PushManager.supportedContentEncodings
                                || ['aesgcm'])[0];
    console.log('Received PushSubscription: ', JSON.stringify(subscription))
    var endpoint = subscription.endpoint;
    var publicKey = key ? btoa(String.fromCharCode.apply(null,
                               new Uint8Array(key))) : null;
    var authToken = token ? btoa(String.fromCharCode.apply(null,
                                 new Uint8Array(token))) : null;
    return fetch('pretplate.php', {
        method: 'POST',
        body: JSON.stringify({
          endpoint: subscription.endpoint,
          publicKey: key ? btoa(String.fromCharCode.apply(null,
                               new Uint8Array(key))) : null,
          authToken: token ? btoa(String.fromCharCode.apply(null,
                                 new Uint8Array(token))) : null,
          contentEncoding,
        })
    }).then(() => subscription);
}
```

Naravno, korisnik može posjećivati našu aplikaciju koristeći više različitih uređaja, točnije, više različitih preglednika. Zbog toga što korisniku želimo poslati push obavijest na svaki uređaj s kojeg je on posjetio našu stranicu, za svaki preglednik je potrebno pribaviti i pohraniti dozvolu. Opisani dio se postiže prosljeđivanjem podataka o pretplati poslužiteljskoj skripti *pretplate.php* koja se sastoji od niže navedenog koda.

```
<?php
require_once 'db.class.php';
require __DIR__ . '/vendor/autoload.php';
use Minishlink\WebPush\WebPush;
```

```
use Minishlink\WebPush\Subscription;
session_start();

$subscription = json_decode(file_get_contents('php://input'), true);
if (!isset($subscription['endpoint'])) {
  echo 'Error: not a subscription';
  return;
}

$endpoint = $subscription['endpoint'];
$auth = $subscription['authToken'];
$p256dh = $subscription['publicKey'];
$id_studenta = $_SESSION['id_stud'];

    try {
        $db = DB::getConnection();
        $st = $db->prepare("INSERT INTO pretplate(id_studenta, endpoint,
                                                  p256dh, auth)
                            VALUES(:id_studenta, :endpoint, :p256dh, :auth)");
        $st->execute(array('id_studenta' => $id_studenta,
                           'endpoint' => $endpoint,
                           'p256dh'=> $p256dh, 'auth' => $auth));
      }
      catch( PDOException $e ) { exit( 'PDO error ' . $e->getMessage() ); }
?>
```

U bazi podataka *Studenti* na poslužitelju kreiramo novu tablicu pod nazivom *pretplate*. U tu tablicu ćemo spremati podatke o pretplatama korisnika na push obavijesti, pa stoga tablica ima sljedeće atribute: *id_studenta*, *p256dh*, *endpoint* te *auth*.

Prilikom prijave studenta, u *$_SESSION['id_stud']* se pohrani identifikacijski broj studenta koji se dohvaća u trenutku spremanja pretplate u bazu. Jedan redak tablice *pretplate* predstavlja detalje o pretplati studenta čiji je identifikacijski broj jednak atributu *id_studenta*. Ukoliko postoji više redaka s istom vrijednosti atributa *id_studenta*, to znači da se student prijavio u aplikaciju s više različitih uređaja.

Kôd koji smo dodali skripti *rezultati.html* će se pokrenuti prilikom svakog posjeta studenta aplikaciji. To znači da će se svaki puta provjeriti je li pribavljena dozvola za prikazivanje obavijesti tom korisniku na uređaju s kojeg trenutno posjećuje aplikaciju. Ako jest, ne kreira se nova pretplata, ako nije, nova pretplata se kreira i sprema u bazu podataka.

Primjetimo da u poslužiteljske skripte moramo uključiti Web Push PHP bliblioteku navodeći sljedeći kôd na početak istih (*server.php, pretplate.php*).

```
<?php
require __DIR__ . '/vendor/autoload.php';
use Minishlink\WebPush\WebPush;
use Minishlink\WebPush\Subscription;
?>
```

Slanje push obavijesti događa se u onom trenutku kada se na u bazu podataka *Studenti* na poslužitelju pohrane novi rezultati za nekog studenta. Tada se dohvaćaju pretplate tog studenta za sve uređaje na kojima se prijavio u aplikaciju te se na sve njih šalju obavijesti s odgovarajućom porukom. Skripta *server.php* brine o spremanju podataka o rezultatima u bazu podataka, pa u nju dodajemo kôd koji omogućava slanje push obavijesti studentima.

```
<?php
try {
    $db = DB::getConnection();
    $st1 = $db->prepare('SELECT * FROM pretplate
                                 WHERE id_studenta=:id_studenta');
    $st1->execute(array('id_studenta' => $id_studenta));
    }
  catch( PDOException $e ) { exit( 'PDO error ' . $e->getMessage() ); }

  while($row1 = $st1->fetch()) {
    $message['w'] = "yas1";
    $subscription = Subscription::create(
    [
      "endpoint" => $row1['endpoint'],
      "keys" => [
        'p256dh' => $row1['p256dh'],
        'auth' => $row1['auth']
      ]
    ]);

    $auth = array(
      'VAPID' => array(
          'subject' => 'https://rp2.studenti.math.hr/~manekic/pwa/',
          'publicKey' => file_get_contents(__DIR__ . '/keys/public_key.txt'),
```

```
            'privateKey' => file_get_contents(__DIR__ . '/keys/private_key.txt'),
        ),
    );

    $webPush = new WebPush($auth);
      $res = $webPush->sendNotification(
        $subscription,
        $message['ime']." , upisani su Vam novi bodovi iz kolegija"
        .$message['ime_kolegija']. " Iz ".$message['podrucje'] .
        " ste dobili " .$bodovi ." bodova."
      );

    foreach ($webPush->flush() as $report) {
      $endpoint = $report->getRequest()->getUri()->__toString();

      if ($report->isSuccess())
          echo "[v] Message sent successfully for subscription {$endpoint}.";
      else
          echo "[x] Message failed to sent for subscription {$endpoint}:
                {$report->getReason()}";
    }
}
?>
```

Standardnim manipulacijama bazom dohvatimo podatke o studentu i kolegiju (ime i prezime studenta te naziv kolegija) kako bismo mogli presonalizirati push obavijesti.

Preostalo je uslužnu skriptu pretplatiti na događaj *push*. Dodavanje koda

```
self.addEventListener('push', function (event) {
  if (!(self.Notification && self.Notification.permission === 'granted')) {
    return;
  }

  const sendNotification = body => {
    const title = "Poruka nakon unosa rezultata";
    return self.registration.showNotification(title, {
      body,
    });
  };
```

```
  if (event.data) {
    const message = event.data.text();
    event.waitUntil(sendNotification(message));
  }
});
```

u uslužnu skriptu uzrokuje da ona započne osluškivati na događaje slanja push obavijesti.

Još samo jedan korak nas dijeli do preobrazbe naše web-aplikacije u progresivnu web-aplikaciju. Omogućavanjem pozadinske sinkronizacije i push obavijesti osigurali smo našim korisnicima mnogo.

1. Administrator više ne mora imati osiguranu stabilnu internetsku vezu želi li unositi rezultate ispita.
2. Studenti više ne moraju konstantno osvježavati stranicu kako bi saznali jesu li došli novi rezultati.

U svakom slučaju, aplikaciju smo učinili, u punom smislu riječi, *user friendly* te smo korisničke potrebe stavili na prvo mjesto.

Poglavlje 7

Manifest

Preostalo je omogućiti korisnicima da našu web-aplikaciju preuzmu na svoje mobilne uređaje kako ne bi morali svaki puta otvarati svoj preglednik u želji da vide informacije koje sadrži aplikacija. Kako bismo to postigli, potrebno je ispuniti sljedeće tri stavke:

1. Registrirati uslužnu skriptu;
2. Kreirati manifest datoteku za web-aplikaciju;
3. Dodati poveznicu za manifest u web-aplikaciji.

Registrirana uslužna skripta koja ima kontrolu nad web-aplikacijom već postoji, pa je potrebno omogućiti preostale dvije stavke.

Manifest je jednostavna JSON datoteka koja opisuje kako se web-aplikacija pokreće, kako izgleda te kako se ponaša. Unutar projekta kreiramo novu skriptu naziva *manifest.json* te joj dodamo sljedeći sadržaj:

```
{
  "short_name": "Rezultati ispita",
  "name": "Praaćenje rezultata na ispitima",
  "start_url": "https://rp2.studenti.math.hr/~manekic/pwa/",
  "display": "standalone",
  "icons": [
    {
      "src": "rsz_math.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
```

```
      "src": "rsz_1math.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ]
}
```

Unutar HTML tag-a *head* svih klijentskih skripti dodajemo

```
<link rel="manifest" href="manifest.json">
```

kako bismo pregledniku dali do znanja da je manifest datoteka dostupna za stranice naše aplikacije.

7.1 Atributi manifest datoteke

Da bi se otvorio prozor za instalaciju web-aplikacije na mobilni uređaj, manifest datoteka mora sadržavati sljedeće atribute:

- *name* i/ili *short_name*: ime i/ili skraćeno ime aplikacije. *name* označava puno ime aplikacije te se koristi onda kada ima dovoljno mjesta za prikaz punog imena. U slučaju kada nema dovoljno mjesta za prikaz punog imena aplikacije, pomoću atributa *short_name*, prikazuje se skraćeno ime iste. Navedeni atributi se prikazuju na prozoru za instalaciju aplikacije ili na početnom zaslonu mobilnog uređaja, odmah pored ikone aplikacije.

- *start_url*: URL koji se treba otvoriti kada korisnik klikne na ikonu aplikacije. To bi trebala biti početna stranica naše aplikacije.

- *icons*: polje koje sadrži jedan ili više objekata, pri čemu svaki od njih opisuje ikonu koju web-aplikacija može koristiti. Svaki od objekata ima svoje atribute: *src* (URL slike), *type* (tip datoteke), *sizes* (dimenzije slika izražene u pixelima). Za instalacijski prozor aplikacije, manifest datoteka mora sadržavati barem jednu ikonu dimenzija 144 px $\times$ 144 px. Što se tiče ikone koja se prikazuje na zaslonu mobilnog uređaja, preporuča se dimenzija barem 192 px $\times$ 192 px te 512 px $\times$ 512 px koja odgovara većini uređaja.

- *display* može poprimiti tri različite vrijednosti: *browser* (otvaranje aplikacije u pregledniku), *standalone* (otvaranje aplikacije bez značajki preglednika, primjerice bez adresne trake), *fullscreen* (otvaranje aplikacije bez ikakvih značajki uređaja i preglednika).

Kao dodatni atributi manifest datoteke mogu se navesti: *description* (opis aplikacije), *orientation* (orijentacija zaslona koja se mora primijeniti prilikom otvaranja aplikacije) i slično.

7.2 Kada će se prikazati instalacijski prozor?

Kada preglednik ustanovi da je web-aplikacija pogodna za instalaciju i da korisnik koji je trenutno posjećuje ima interes za tu aplikaciju (primjerice, da bi volio imati web-aplikaciju na početnom zaslonu svog mobilnog uređaja), prikazat će instalacijski prozor.

Instalacijski prozor će se prikazati samo ako su navedene stavke zadovoljene:

1. Aplikacija se poslužuje preko sigurne veze (HTTPS);
2. Aplikacija ima registriranu uslužnu skriptu;
3. Manifest aplikacije sadrži sve potrebne atribute (navedene u odjeljku 7.1).

Dodatno, korisnik web-aplikaciju mora posjetiti nekoliko puta prije nego što mu se prikaže instalacijski prozor (zbog toga što preglednik prvo mora ustanoviti da korisnik ima interes za aplikaciju). Nakon što projektu dodamo datoteku *manifest.json* te svi uvjeti budu zadovoljeni, našim korisnicima će se na mobilnim uređajima prikazati skočni prozor kao na slici 7.1.

Slika 7.1: Skočni prozor za instalaciju web-aplikacije na mobilni uređaj

Bibliografija

[1] T. Ater, *Building Progressive Web Apps*, O'Reilly Media, 2017, posjećeno u kolovozu 2019.

[2] Composer, https://getcomposer.org/, Alat za upravljanje ovisnostima aplikacija pisanih u PHP-u, posjećeno u kolovozu 2019.

[3] Google, *Podrška za razvoj progresivnih web-aplikacija*, https://developers.google.com/web/progressive-web-apps/, posjećeno u kolovozu 2019.

[4] Mozilla, *Progresivne web-aplikacije*, https://developer.mozilla.org/en-US/docs/Web/Progressive, posjećeno u kolovozu 2019.

[5] Netlify, *International Service Worker Caching Awareness Day*, https://www.netlify.com/blog/2018/09/21/international-service-worker-caching-awareness-day, posjećeno u kolovozu 2019.

[6] Web-Push-PHP, https://github.com/web-push-libs/web-push-php, Biblioteka za slanje push obavijesti pomoću PHP-a, posjećeno u kolovozu 2019.

Sažetak

U ovom radu smo s teorijskog stajališta objasnili što su to progresivne web-aplikacije (PWA) te smo na konkretnom primjeru pokazali kako web-aplikaciju pretvoriti u progresivnu.

Na početku smo objasnili što su uslužne skripte, najvažniji dio PWA. One omogućavaju trenutno pokretanje PWA, bez obzira na veličinu sadržaja aplikacije te na stanje mreže. To se postiže predmemoriranjem ključnih resursa pomoću kojeg možemo eliminirati ovisnost o mreži i na taj način zajamčiti odaziv i pouzdan doživljaj za korisnike. Naveli sam neke od najčešće korištenih obrazaca predmemoriranja te objasnili što je to IndexedDB (lokalna baza podataka unutar preglednika) i kako se njome koristiti.

Dvije nove funkcionalnosti koje smo prikazali su tzv. push obavijesti i pozadinska sinkronizacija. One omogućavaju aplikaciji da, čak i nakon što je korisnik napusti, ona nastavi komunikaciju s istim te da sve potrebne informacije zaista budu dostavljene poslužitelju, čak i u slučaju privremenog gubitka internetske veze.

Pisanjem manifest datoteke, postigli smo da se PWA može instalirati na korisnikov mobilni uređaj te smjestiti na početni zaslon. Instalacija se odvija bez potrebe za trgovinom aplikacija, direktno s web-lokacije na kojoj se ta web-aplikacija prikazuje.

U svrhu demonstracije da aplikacija korektno radi u stanju kada ima pristup mreži i kada nema, uveli smo identifikatore stanja koji reagiraju na događaje promjene stanja mreže i to prikazuju.

Summary

In this thesis, we have explained from the theoretical point of view what are progressive web applications (PWAs) and have shown in a concrete example how to transform a web application into a progressive web application.

At the beginning we explained what are service workers, the most important part of PWA. They enable instant launch of PWAs, regardless of the size of the application's content and network status. That is achieved by caching key resources. Caching allows us to eliminate network dependency and thus ensures a responsive and trustworthy user experience. We explained some of the most frequently used caching patterns. Also, we explained what is IndexedDB (local database inside the browser) and how to use it.

Two new features we have highlighted are push notifications and background sync. Push notifications allow the app to continue communicating with the user even after he leaves the app. Background sync ensures that all necessary data is actually delivered to the server even in the event of temporary Internet connection loss.

By writing a manifest file, we have achieved that the PWA can be installed on the user's mobile device and placed on the home screen. Installation takes place without the user ever visiting the app store, directly from the site where that web app is running.

For the purpose of demonstrating that the application works correctly regerdless of whether the user has network access or not, we have introduced status identifiers that respond to network changing events and display it.

Životopis

Rođena sam 25. kolovoza 1994. godine u Bjelovaru. Po završetku osnovne škole u Čazmi, svoje školovanje sam odlučila nastaviti u Općoj gimnaziji, također u Čazmi. U osnovnoj školi sam sudjelovala na školskim i županijskim natjecanjima iz biologije, kemije i fizike. U srednjoj školi sam išla na natjecanja iz informatike, područje: Osnove informatike.

Nakon završetka srednje škole, 2013. godine sam upisala Preddiplomski sveučilišni studij Matematike na Prirodoslovno matematičkom fakultetu u Zagrebu. Zatim sam 2017. godine sam upisala Diplomski sveučilišni studij, smjer: Računarstvo i matematika.

U razdoblju između polaganja posljednjih ispita i obrane diplomskog rada, kao student sam se zaposlila u IT sektoru firme AVL-AST.