SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1827

IZGRADNJA UDOMLJENIKA PRIMJENOM OBLIKOVNOG OBRASCA MODEL-PRIKAZ-UPRAVLJAČ

Goran Petrović

Zagreb, prosinac 2009.

OVDJE UMETNUTI ORIGINALNI ZADATAK

ZAHVALA:
Mentoru prof. dr. sc. Siniši Srbljiću na izvrsnoj prilici i vodstvu.
Roditeljima koji su vjerovali u mene i podržavali me.

Sadržaj

Uvod 1

1. Izgradnja primjenskih sustava primjenom paradigme model-prikaz-upravljač 3

1.1. Paradigma model-prikaz-upravljač 3
1.1.1. Model 4
1.1.2. Prikaz 6
1.1.3. Upravljač 6
1.2. Sinkronizacija konteksta pretplaćivanjem 6
1.3. Primjer uporabe paradigme model-prikaz-upravljač 7

2. Web tehnologije i sustavi 9

2.1. Document Object Model 9
2.2. *JavaScript* programski jezik 13
2.2.1. *JavaScript* u web preglednicima 14
2.2.2. *JavaScript* objektna notacija 16
2.3. XML 18
2.4. HTTP protokl 19
2.4.1. HTTP zahtjev 19
2.4.2. HTTP odgovor 21
2.5. AJAX 22
2.6. Google Gadget sustav 22
2.6.1. XML struktura 24
2.6.2. *OpenSocial* okruženje 25
2.7. Google App Engine sustav 26
2.7.1. Java Runtime okruženje 26
2.7.2. Java Servlet tehnologija 27
2.7.3. Java Data Objects 28

3. Udomljenik za izradu korisničkog sučelja 29

3.1. Uloga i funkcija oblikovanog udomljenika 29
3.1.1. Funkcija udomljenika u razvoju drugih udomljenika 29

3.2. Primjena udomljenika . . . 31
3.2.1. Radna površina. . . . 31
3.2.2. Uređivanje atributa HTML elemenata . . . 32
3.2.3. Uređivanje `style` atributa . . . 33

4. Arhitektura udomljenika za oblikovanje korisničkog sučelja 35
4.1. Arhitektura korisničkog sučelja . . . 35
4.2. Arhitektura repozitorija na poslužitelju . . . 37
4.3. Povezivanje klijenta i poslužitelja . . . 39

5. Programsko ostvarenje korisničkog sučelja 41
5.1. Prezentacijski sloj udomljenika . . . 41
5.2. Sinkronizacija prikaza . . . 42
5.2.1. Sinkronizacija Toolbox udomljenika . . . 42
5.2.2. Ostvarenje paradigme model-prikaz-upravljač . . . 43
5.3. Posluživanje HTML elemenata . . . 46
5.3.1. Opis HTML elemenata . . . 46
5.3.2. Dinamično dobavljanje opisa elemenata . . . 48

6. Programsko ostvarenje poslužitelja 50
6.1. Struktura sloja za spremanje konteksta . . . 50
6.1.1. Razred *UserKey* . . . 51
6.1.2. Razred *DataEntry* . . . 51
6.1.3. Razred *Singleton* . . . 52
6.1.4. Razred *RepoFetch* . . . 52
6.1.5. Razred *RepoUpload* . . . 53
6.2. Autentikacija korisnika . . . 54
6.2.1. Dodjeljivanje identifikacijskog broja . . . 54
6.2.2. Provjera identifikacijskog broja . . . 54
6.3. Zahtjevi prema poslužitelju . . . 55
6.3.1. Obrada zahtjeva za spremanje konteksta . . . 55
6.3.2. Obrada zahtjeva za dobavljanje konteksta . . . 55
6.3.3. Obrada zahtjeva za brisanje konteksta . . . 55

Zaključak 57

Literatura 58

Uvod

Čovjek XXI. stoljeća sve češće koristi Internet kako za poslovne tako i za osobne potrebe, a Web preglednik se time nametnuo kao jedan od najkorištenijih programa na računalu. Unatoč velikoj složenosti procesa razvijanja primjenskih programa za Internet, popriličnom broju tehnologija i standarda koje je potrebno poznavati te sustavnih nedosljednosti i nekompatibilnosti koje je nužno izbjeći, jedan od ciljeva modernog računalnog inženjerstva je donošenje izrade primjenskog sadržaja krajnjem korisniku koji ne posjeduje obrazovanje u računalnoj znanosti.

Naglim razvojem bežičnih mreža i prijenosnih mrežnih računala korištenih skoro isključivo za pristup Internetu, postalo je važno centralizirati svu potrebnu funkcionalnost i korisnikove podatke tako da su uvijek i svugdje dostupni i organizirani, a da je računalo potrebno za pristup što manje i jeftinije, prebacujući ostvarenje primjenske logike i spremišta podataka na poslužitelj. Osim razvoja poslovnih sustava kao što su uređivači teksta, tabličnih podataka i prezentacija, došlo je i do razvoja jednostavnih primjenskih programa koji se pokreću kao dio web sadržaja stranice, tzv. udomljenika (engl. *gadget*). Udomljenici su postali popularni zbog jednostavne centralizacije skupa podataka i lake prenosivosti sa stranice na stranicu ili pak na radnu površinu računala tj. svoje sveprisutnosti na računalima i mobilnim uređajima bez ikakvih postavljanja sustava.

Sve veće i veće korištenje udomljenika u potrošačkim primjenskim programima i rast složenosti i zahtjeva za sigurnost sustava pridonijelo je razvoju potrošačkih sustava razvijenih oko udomljenika, omogućujući pohranu i obradu podataka na strani poslužitelja, gdje je infrastruktura robusnija, a spremišta podataka sigurna. Performanse klijentskih aplikacija ovise o računalu na kojem se izvode, ali je i vrlo teško kvalitetno ispitati njihvou kakvoću i funkcionalnost, te je i izvođenje bitnih dijelova logike preseljeno na poslužitelj. Sustav *Google App Engine* je sustav za izvođenje primjenskih programa kojem se pristpua mrežnim protokolima, a infrastruktura na kojoj se izvodi je masivan raspodijeljeni sustav održavan od strane

kompanije *Google*.

U ovom radu opisano je oblikovanje *Toolbox* udomljenika za izradu i uređivanje stiliziranih web elemenata i ostvarenje mrežnog repozitorija korištenjem raspodijeljenih sustava te prateće tehnologije uz slikovite primjere. Pri oblikovanju profinjenih mrežnih primjenskih programa, često se uporabljuje paradigma model-prikaz-upravljač kako bi se odvojili logički dijelovi programa. Posebno je prikazana primjena paradigme model-prikaz-upravljač na *Toolbox* udomljenik.

U poglavlju 1 opisana je primjena paradigme model-prikaz-upravljač na izgradnju Google udomljenika. Opisani su osnovi elementi na kojima se paradigma zasniva i njihova međusobna komunikacija. Poglavlje 2 donosi uz primjere pregled i opis osnovnih tehnologija na kojima se zasnivaju mrežni primjenski programi i protokoli kojima komuniciraju. Uloga i funkcija oblikovanog udomljenika, te njegova primjena opisana je u poglavlju 3, dok mu je arhitektura opisana u poglavlju 4. Poglavlja 5 i 6 donose ostvarenje klijenta udomljenika i repozitorija na poslužitelju.

1. Izgradnja primjenskih sustava primjenom paradigme model-prikaz-upravljač

U ovom poglavlju opisana je paradigma model-prikaz-upravljač (engl. *model-view-controller*) i sinkronizacija konteksta pretplaćivanjem. Model-prikaz-upravljač je paradigma koja se koristi u programskom inženjerstvu kako bi se poslovna logika izolirala od logike za spremanje i logike za prikaz podataka. Ovakav način oblikovanja sustava omogućava neovisan i odvojen razvoj, ispitivanje i održavanje svakog dijela sustava. Sinkronizacija konteksta pretplaćivanjem je mehanizam kojim se garantira pravovremenost i valjanost podataka koji se prikazuju korisniku pamćenjem liste slušaoca zainteresiranih za promjene konteksta. Poglavlje 1.1 donosi opis funkcionalnosti modela, prikaza i upravljača, dok se u poglavlju 1.2 iznosi opis sinkronizacije konteksta pretplaćivanjem.

1.1. Paradigma model-prikaz-upravljač

U paradigmi model-prikaz-upravljač modeliranje vanjskog svijeta, korisničke akcije i prezentacija podataka korisniku su eksplicitno odvojeni i rješavaju se sa tri različita objekta, od kojih je svaki specijaliziran za svoju zadaću. Prikaz upravlja grafičkim ili tekstualnim izlazom do nivoa točkastog prikaza koji je dodijeljen primjenskom programu. Upravljač tumači unose miša i tipkovnice korisnika, naređujući modelu i prikazima da se promijene u skladu sa akcijom korisnika. Konačno, model upravlja ponašanjem podataka primjenskog programa, odgovara na upite o svom stanju i instrukcije za promjenu stanja. Formalno odvajanje ovih triju zadaća važna je ideja, koja je prvi put zamišljena u sklopu programskog jezika *Smalltalk*, gdje se osnovno ponašanje može opisati u apstraknim objektima paradigme. Kako

bi se paradigma mogla učinkovito koristiti, nužno je odrediti podjelu posla unutar trojke model-prikaz-upravljač.[3]

Na slici 1.1 prikazan je sustav oblikovan na osnovi paradigme model-prikaz-upravljač koji se sastoji od jednog modela, jednog upravljača i više prikaza, no svako programsko ostvarenje prati osnovna pravila paradigme. Djeluje li korisnik na jedan od prikaza, podiže se događaj kojim prikaz obaviještava upravljač o događanjima (i). Upravljač po potrebi šalje modelu zahtjeve za promjene (ii). Ako se bitni podaci promijene, model signalizira upravljaču kako je došlo do kritične promjene (iii), na što upravljač odgovara slanjem upute za osvježavanje svakom prikazu (iv).

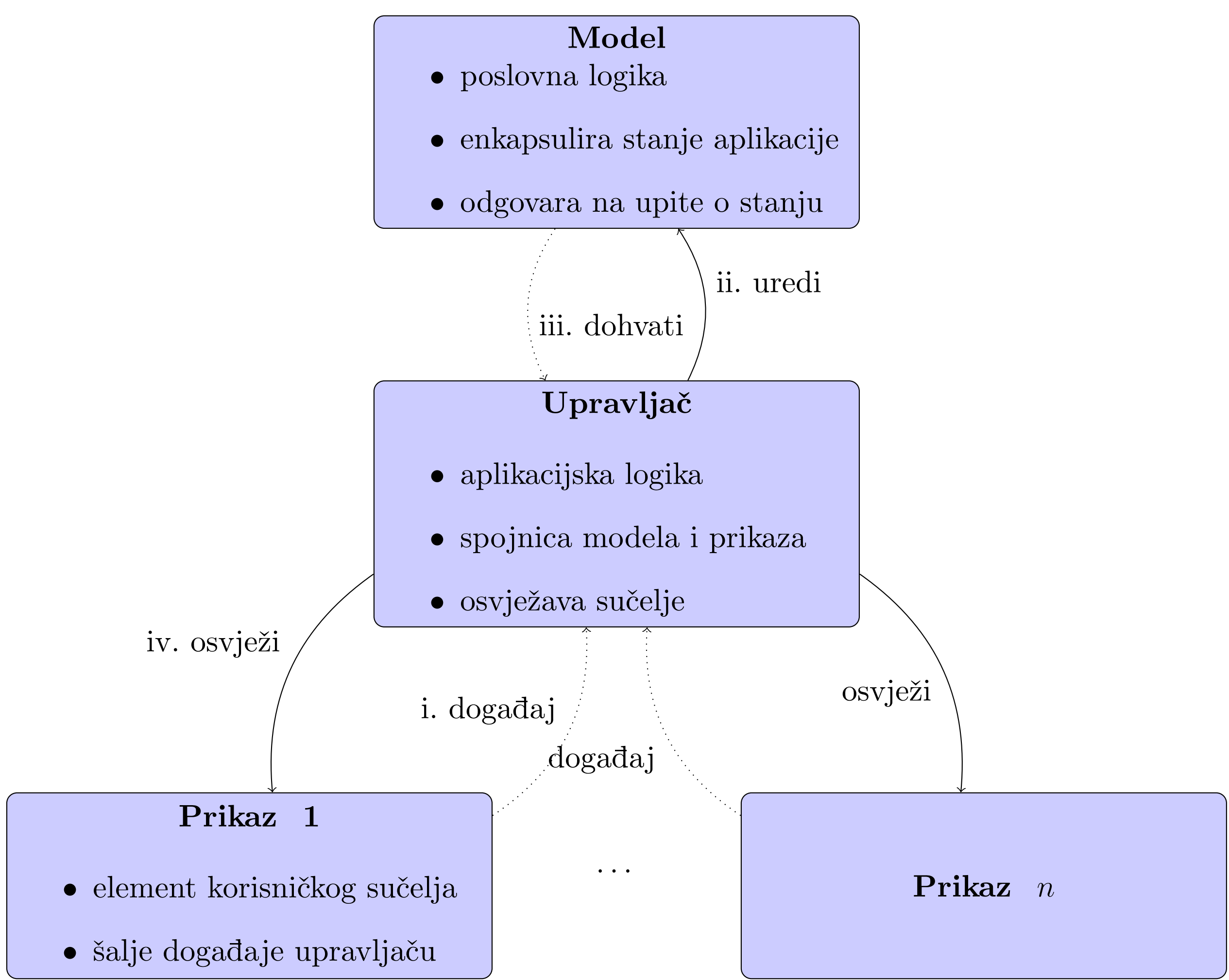
Model služi za predstavljanje svih podataka koje primjenski program koristi. U pozadini modela se nalazi spremište podataka u koje se spremaju podaci u bilo kojem obliku. Spremište ovisi o programskim ostvarenjima, a može biti datoteka na tvrdom disku, *SQL* baza podataka ili pak radna memorija računala. Apstraktnije, model može biti bilo kakav objekt, jednostavan kao jedan broj s pomičnim zarezom ili koncipiran kao cijeli sustav za komunikaciju sa raspodijeljenom bazom podataka.[12]

Prikaz se koristi za predodžbu podataka iz modela u obliku pogodnom za međudjelovanje s korisnikom. Za jedan model moguće je oblikovati više prikaza, od kojih svaki prikazuje kontekstno relevantne podatke i oblike interakcije. Jedan primjer pogleda je Web preglednik koji prikazuje Internet stranicu u grafičkom obliku tranformirajući *HTML* i *CSS* kojim je napisana u oblik bliži korisnicima, dok je za isti sustav drugi pogled sam izvorni kôd koji čini stranicu. Dodatno, ako preglednik prikazuje više Web stranica odjednom, jedan od mogućih pogleda je i popis svih otvorenih stranica koji prikazuje samo naslov stranice.

Upravljačem se koordinira svako međudjelovanje između korisnika i modela podataka. Upravljač indirektno mijenja podatke u modelu na zahtjev korisnika, te osvježava prikaz ako se kontekst promijeni, tj. sadrži ostvarenje primjenske logike. Upravljači su apstrakcije pogodne za postavljanje regresijskih ispitivanja.[12] Primjenom paradigme model-prikaz-upravljač postignut je sinkroniziran i ažuran prikaz podataka i jednostavna replikacija potrebnih prikaza.

1.1.1. Model

Model predstavlja sveukupnost podataka koje primjenski program koristi i logička pravila koja se nad njima nameću, te enkapsulira stanje primjenskog progra-



Slika 1.1. Paradigma model-prikaz-upravljač

ma, tj. teži biti računalna aproksimacija stvarnog procesa. U slučaju rekonstrukcije primjenskog programa, svo stanje se može saznati iz modela, te on pruža način za rad s podacima koje pohranjuje izlažući korisniku skup funkcionalnosti. Modeli najčešće ostvaruju radnu logiku (engl. *business logic*) koja određuje na koji način se obrađuju podaci koji se šalju između korisničkog sučelja i baze podataka. Primjerice, u slučaju da model prati top ljestvicu pjesama, kada se broj glasova promijeni, model mora automatski tražiti osvježavanje svih prikaza top liste kako bi se osigurala točnost i ažurnost.

U slučaju da sustav koristi višemodelni oblik paradigme, svaki logički odvojiv podsustav posjeduje svoj model koji sadrži minimalan podskup sveukupnih informacija koji je vezan za taj podskup kako bi se još više raspodijelio sustav. Oblikovanje više logičkih odvojenih modela za primjenski program omogućava lakšu zamjenu dijelova sustava, ostvarivanje novih pristupa istom problemu za različite

okoline te ispitivanje sustava. Promjenom dijela sustava, njegovo održavanje i ispitivanje je iznimno otežano u slučaju neraspodijeljenosti te je zato vrlo bitno ispitati međuovisnosti modela.

1.1.2. Prikaz

Prikaz je odgovoran za prikazivanje podataka iz modela na korisniku uporabljiv način, tj. za enkapsulaciju prezentacijske logike. Prikazi ne bi smjeli sadržavati nikakvu poslovnu logiku ili komunicirati sa bazom podataka; svu aplikacijsku logiku mora sadržavati upravljač. Prikaz smije samo iscrtavati korisničko sučelje koristeći podatke koje mu preda upravljač, te na akciju korisnika proslijediti upravljaču događaj.

Za isti primjenski program moguće je ostvariti više prikaza, koji svaki na svoj način generira element korisničkog sučelja i upućuje upravljaču događaje koje zaprima od korisnika. U razvoju grafičkih alata, primjerice, ovaj koncept se očituje u grafičkom prikazu same slike koja se uređuje i prikazu koji sadrži popis i nazive slojeva (engl. *layer*) koji postoje na slici.

1.1.3. Upravljač

Upravljači obrađuju zahtjeve korisnika i interakciju, te izvršavaju primjerenu aplikacijsku logiku, najčešće obavještavajući prikaze da se osvježe novim podacima ili šaljući modelu zahtjeve za promjenama. Upravljač služi kao ljepilo između podataka i korisničkog sučelja. Na svaki zahtjev pogleda, upravljač zatraži od modela provođenje traženih promjena, te obavještava sve poglede o nastalim promjenama; logika koju ostvaruje nema veze sa semantikom modela već s pravilnim radom primjenskog programa. U nekim ostvarenjima, kroz upravljač se koriste grafovi ovisnosti u svrhu automatizacije osvježavanja prikaza kada se podaci u modelu promijene.

1.2. Sinkronizacija konteksta pretplaćivanjem

Garancija ažurnosti podataka iznimno je važna u današnjim interaktivnim sustavima, te je zbog toga razvijeno više sinkronizacijskih sustava koji osiguravaju točnost i pravovremenost podataka. Raspodijeljeni primjenski programi pružaju velikom broju korisnika istovremeno korištenje iste usluge (npr. *Google Wave*), u

kojoj svaki korisnik ima pristup sebi relevantnim podacima. Korisnici mijenjaju sadržaje svojih podataka, dodaju nove i brišu ili dijele postojeće. Jedan takav sustav mora pravovremeno obavještavati korisnike o tome što se događa sa podacima kojima oni imaju pristup kako korisnici ne bi otvarali podatke koji su već bili izbrisani ili mijenjali već izmjenjene podatke te bi se prijašnje izmjene izgubile.

U višeprikaznom sustavu s jednim upravljačem moguće je pretplatiti prikaze na novosti koje šalje upravljač. Svaki put kada upravljač izmjeni kontekst modela, on obaviještava svaki pretplaćeni prikaz da se treba osvježiti jer postoji mogućnost da nije ažuran, tj. da je registrirana promjena konteksta utjecala na onaj podskup modela koji taj pogled prikazuje.

Kako bi se prikazi mogli pretplatiti na obavijesti, upravljač mora omogućavati pretplate, tj. pamtiti listu svih slušatelja (engl. *listeners*). Za svaku promjenu konteksta, upravljač svakom slušatelju kojeg poznaje tj. koji je kod njega pretplaćen šalje obavijest da je model promijenjen te da se prikaz treba osvježiti. Svaki prikaz koji želi primati obavijesti mora se pretplatiti kod meritornog upravljača. tj. zahtjevati da ga se doda u listu slušatelja. Registriranim prikazima osigurana je točnost i pravovremenost podataka u granicama brzine protoka podataka.

1.3. Primjer uporabe paradigme model-prikaz-upravljač

Primjena paradigme model-prika-upravljač često se susreće u današnjim tehnologijama kao što su *ASP.NET MVC Framework* ili *Django* – okvir za razvoj web primjenskih programa u programskom jeziku *Python*, no prisutna je i primjenskim programima koji se svakodnevno koriste i u programerski nestručnim krugovima, kao što su primjenski programi za uređivanje slika. Osnovan prikaz takvih programa je, naravno, sam prikaz slike koja se uređuje, no alternativni prikazi mogu biti popis postojećih slojeva ili pak njihov sadržaj. Na slici 1.2 prikazana su tri prikaza slike u primjenskom programu za uređivanje slika. Prikaz sa lijeve strane je binarni zapis datoteke na tvrdom disku tj. u radnoj memoriji računala koji ujedno predstavlja model za sliku. Prikaz datoteke u datotečnom sustavu varira od sustava do sustava, no svako ostvarenje zapisuje polje bajtova koje predstavlja sliku u zadanom formatu, a na sličan se način podaci prikazuju u radnoj memoriji. Takav prikaz je pogodan računalu, no za korisnika koji želi sliku predočiti ili urediti je sasvim beskorisan. Programi za predočavanje slika iz niza bajtova, koristeći poznate

Slika 1.2. Prikazi u programu za uređivanje slika

filtere za tumačenje ovisno o formatu slike, iscrtavaju sliku na zaslon, time predstavljajući drugi prikaz istih podataka, tj. onaj srednji na slici 1.2, gdje korisnik može vidjeti sliku. Prikaz desno na slici predočava treći pogled na iste podatke, točnije popis i sadržaj slojeva od kojih se slika sastoji.

2. Web tehnologije i sustavi

U ovom poglavlju iznesen je kratak pregled osnovnih Web tehnologija.

Poglavlje 2.4 donosi osnove *HTTP* protokola koji je glavni komunikacijski protokol aplikacijske razine na Internet. U poglavlju 2.3 opisane su osnove *XML* jezika koji služi za opisivanje i oblikovanje podataka, dok poglavlje 2.2 ukratko opisuje programski jezik *JavaScript* koji se koristi za povezivanje sa logičkom strukturom Web stranice unutar preglednika. Sustav na kojem se izvršavaju Google udomljenici opisan je u poglavlju 2.6, a poglavlje 2.7 donosi opis *Google App Engine* sustava koji omogućuje trajno pokretanje primjenskih programa korisnika na Google infrastrukturi.

2.1. Document Object Model

Document Object Model je platformski i jezično neovisan standard za prikazivanje i međudjelovanje sa objektima unutar *HTML*, *XHTML* i *XML* dokumenata. Elementima *DOM* modela, kao što je *HTML* element može se pristupiti kroz sintaksu programskog jezika koji se koristi. Javna sučelja *DOM* modela specificirana su u *API* sučelju.[14]

DOM omogućava programski pristup elementima Web stranice, ali i prozoru preglednika u kojem se prikazuje stranica kako bi se programski mogla promijeniti trenutna adresa stranice koja se prikazuje ili pak osvježiti trenutna stranica. Dodatno, omogućava se otvaranje novih iskačućih prozora (engl. *popup*). Postoje četiri glavna objekta za dinamičko upravljanje stranicom: `document`, `window`, `location` i `history`.[17] Kad preglednik zaprimi odgovor od nekog poslužitelja koji sadrži *HTML* oblik stranice, on raščlanjuje elemente i stvara stablo svih elemenata ukorijenjeno u `document` objektu. Dok svaki preglednik obavlja gramatičko raščlanjivanje na svoj način, od svakog se očekuje da se drži propisanih standarda, no promatrajući stvorena stabla, lako se uviđa da su vrlo često različita za isti *HTML*

kôd, što uzrokuje probleme usaglašenost raznih primjenskih programa, ali je i dio političke borbe na tržištu preglednika i operacijskih sustava.

U ispisu 2.1 dan je jednostavan *HTML* kôd koji preglednik raščlanjuje na elemente i gradi *DOM* stablo. Svaki element predstavljen je jednim čvorom, a svi elementi koji su unutar drugih elemenata prikazani su kao djeca tog elementa. Na slici 2.1 prikazano je kondenzirano stablo koje preglednik gradi raščlanjivanjem danog kôda. Plavom bojom prikazani su sagrađeni elementi, dok su listovi, tj. krajnje tekstualne vrijednosti dane zelenim čvorovima. Prema W3 standardu, za prazne znakove ili oznake novih linija $\backslash n$ se također izrađuju čvorovi, no tog standarda se ne drže svi (npr. *Microsoft*) zato što smatraju da se stvara zagušavanje stabla nekorisnim čvorovima koji ne nose značenje, ali time se uzrokuje nekompatibilnost. Takvi su čvorovi na slici ispušteni radi preglednosti. Slika 2.2 donosi prikaz danog kôda u web pregledniku.

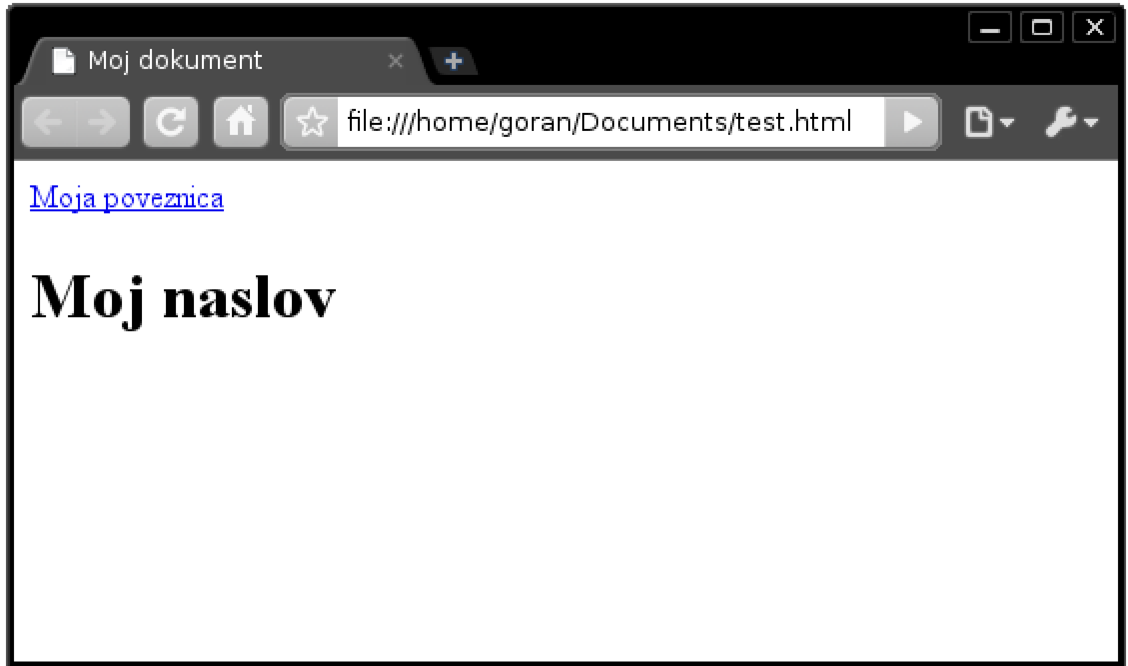
```
1  <html>
2    <head>
3      <title>Moj dokument</title>
4    </head>
5    <body>
6      <a href="http://www.fer.hr/">Moja poveznica</a>
7      <h1>Moj naslov</h1>
8    </body>
9  </html>
```

Ispis 2.1. Primjer *HTML* dokumenta

Prozor preglednika predstavljen je objektom `window`. Tipično, `window` sadrži `document` kao dijete i pruža pristup `window.navigator` i `window.screen` objektima za manipuliranje okruženja samog preglednika te osigurava nekoliko posebnih metoda za dobavljanje svojstava preglednika, kao što je veličina prozora, veličina zaslona ili koordinate na kojima se iscrtava sadržaj. Dodatno, objekt `window` pruža informacije o samom pregledniku, njegovo ime, proizvođača i verziju, trenutni jezik i platformu koji mogu biti korisni primjenskom programu. Sučelje za međujelovanje sa DOM modelom omogućava pomoću objekta `document` dobavljanje elemenata stabla, pretraživanje stabla prema raznim kriterijima te dodavanje, brisanje i izmjenu elemenata i njihovih atributa, što direktno omogućava dinamična korisnička sučelja koja mogu brzo reagirati na korisnikove akcije i uzrokovati događaje (engl. *event*) nad modelom. Događaji se obrađuju *JavaScript* funkcijama koje pregled-

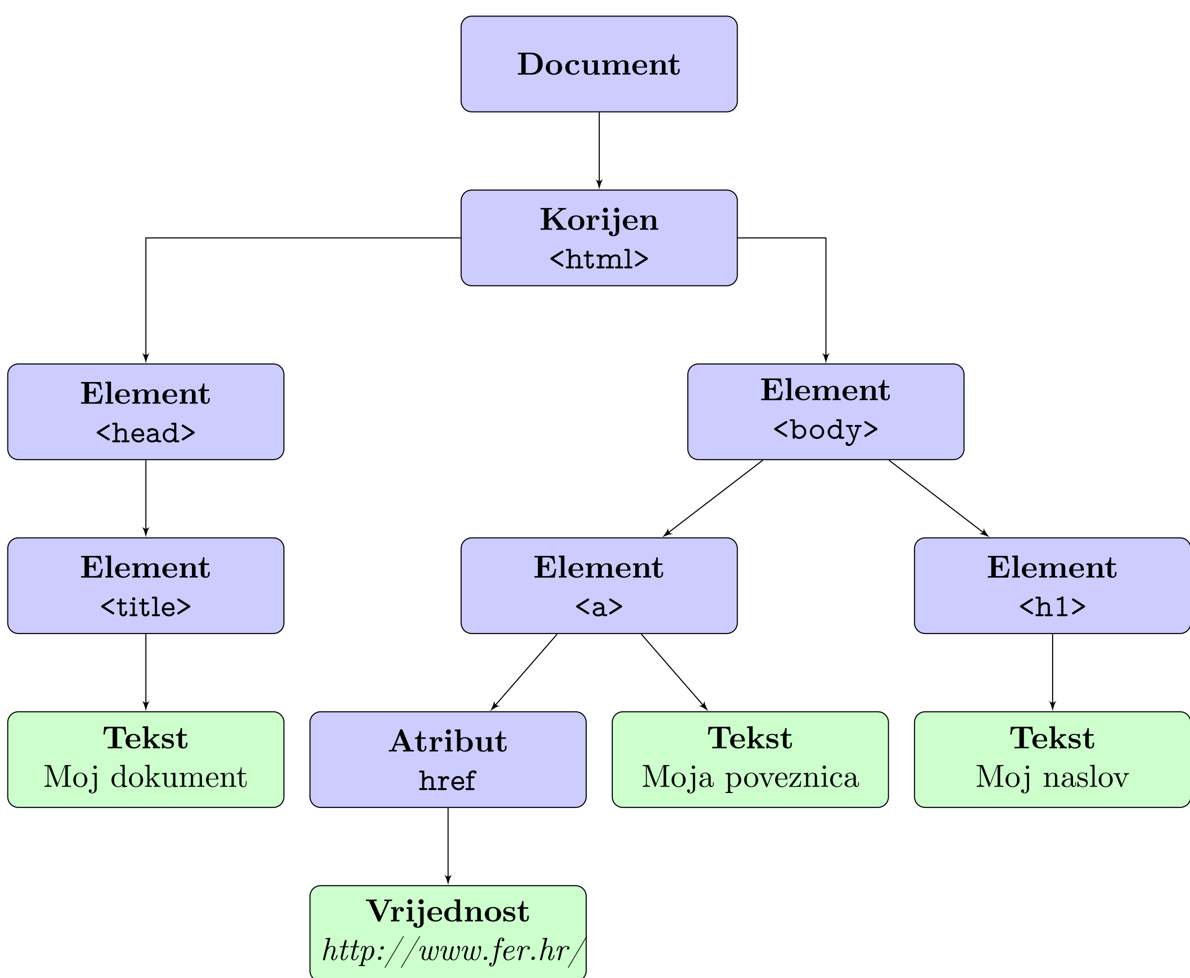
nik poziva kad se dogode određeni događaji. Moguće je dohvatiti element, ali osim mijenjanja, moguće mu je dodavati nove elemente – djecu ili brisati postojeće ili pomicati element po stablu.

Kako bi se moglo upravljati adresom koju preglednik prikazuje, preglednik omogućava pristup objektu `window.location` kojim se može dobaviti trenutni protokol (npr. *http, https, ftp, ssh, ...*), adresa, broj porta stranice te upitni niz (engl. *query string*), dio adrese nakon znaka "?". Dodatno, objekt `window.location` nudi promjenu trenutne adrese, tj. preusmjerenje trenutnog prozora preglednika na neku drugu web stranicu.



Slika 2.2. Prikaz *HTML* dokumenta u web pregledniku

Preglednici pružaju korisniku navigaciju prošlosti (engl. *history*) pregledavanja korištenjem *naprijed* i *natrag* funkcionalnosti, no ista funkcionalnost može se postići programski, pomoću posebnog objekta `window.history` koji omogućava programsko upravljanje navigacijom, tj. kretanje unaprijed i unatrag za proizvoljan broj koraka. Upravo je ovaj objekt instrumentalan u ostvarenjima klijentskih razvojnih okvira (npr. *Google Web Toolkit*), koji omogućavaju programsko stvaranje virtualne prošlosti jedne dinamične stranice gdje se svakom stanju stranice pridaje opis tog stanja i globalna funkcija rekonstrukcije.



Slika 2.1. Osnovno DOM stablo nakon gramatičkog raščlanjivanja

2.2. *JavaScript* programski jezik

JavaScript programski jezik je jedan od dijalekata *ECMAScript* standardnog koji uz osnove sadrži i standardne biblioteke.[5] *JavaScript* je objektno orijentirani skriptni jezik koji se koristi za programsko upravljanje objektima unutar klijentovog primjenskog programa i ostalih programa. Najčešće se koristi u obliku klijentske *JavaScript* koja je ostvarena kao komponenta web preglednika, omogućavajući razvoj poboljšanog korisničkog sučelja i dinamičnih primjenskih programa. Osnovne karakteristike *JavaScript* jezika su dinamičnost, prototipno nasljeđivanje, funkcionalnost i slabi tipovi varijabli. JavaScript jezik je oblikovan da izgleda kao *Java* ali da bude jednostaviji za rad ljudima koji nisu programeri.[8]

JavaScript programski jezik podržava dinamične tipove podataka, tj. varijable se ne deklariraju po tipovima nego se tip određuje dinamički, kada se varijabli pridijeli vrijednost. Naknadno pridjeljivanje vrijednosti istoj varijabli potencijalno mijenja njen tip. *JavaScript* objekti su asocijativna polja proširena prototipovima. Svojstva objekata se mogu dinamički dodavati, mijenjati i brisati za vrijeme izvođenja. Funkcije u *JavaScript* jeziku igraju važnu ulogu; one su objekti i imaju svojstva koja se isto tako mogu dinamično mijenjati, te ih se može slati kao argumente. Unutarnje funkcije, tj. funkcije definirane unutar drugih funkcija, stvaraju se svaki put kad se pozove vanjska funkcija, dok varijable vanjske funkcije za taj poziv nastave postojati dok god unutarnja funkcija postoji, čak i nakon što vanjska funkcija završi izvođenje. Taj mehanizam se naziva *closures*.[8]

```
1  function pocetak() {
2    print('Dobar dan, svijete!');
3  }
4  pocetak();
```

Ispis 2.2. Primjer *JavaScript* programa u interaktivnoj okolini

Primjer jednostavnog *JavaScript* programa za interaktivni prevodioc dan je u ispisu 2.2.

JavaScript koristi prototip za nasljeđivanje, a ne razrede, iako je koristeći prototipove moguće simulirati večinu svojstava razrednog nasljeđivanja. Funkcije se dodatno koriste kao konstruktori objekata na način da se prije poziva funkcije umetne ključna riječ `new` (npr. `varijabla = new MojKonstruktor()`). Konstruktorovo `prototype` svojstvo određuje objekt koji se koristi za unutarnji prototip novog

objekta. *JavaScript* ne razlikuje funkcije od metoda, tj. ako se funkcija pozove kao metoda nad objektom, funkcijina lokalna ključna riječ `this` se veže za taj objekt za vrijeme poziva. Funkcijama se može predati neodređen broj parametara koji se mogu dohvatiti pomoću lokalnog `arguments` objekta. *JavaScript* jezik podržava vrlo snažan mehanizam za regularne izraze slično kao *Perl* pomoću integrairanog `RegEx` objekta.

2.2.1. *JavaScript* u web preglednicima

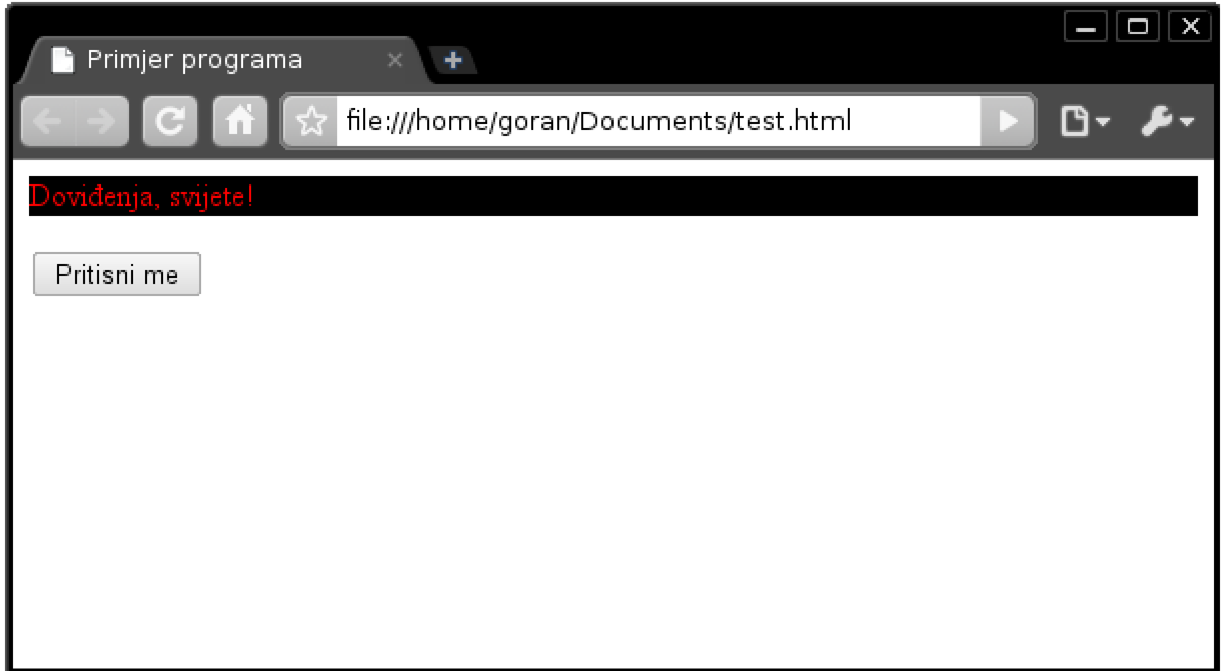
Primarna uporaba *JavaScript* jezika je pisanje funkcionalnosti ugrađene ili uključene u *HTML* stranice i interakcija sa straničinim modelom *DOM*. Jednostavni primjeri korištenja *JavaScript* programskog jezika sadržavaju otvaranje novih prozora kojima su veličina, pozicija i sadržaj programski promjenjivi, ovjeru Web obrazaca i izmjenu slika kada se kursor miša nalazi preko njih.

Kako se *JavaScript* može izvršavati na klijentu, unutar Web preglednika za razliku od izvršavanja na udaljenom web poslužitelju, može brže odgovarati na korisnikove zahtjeve čineći primjenski program dinamičnijim. Nadalje, *JavaScript* može pratiti događaje koje sam *HTML* ne može, kao što je pritiskanje pojedinih tipki na tipkovnici. Mnogi primjenski programi koriste *JavaScript* kako bi logiku korisničkog sučelja donijeli na klijent, a od poslužitelja isključivo zahtjevali potrebne podatke.

JavaScript funkcionalnost se ugrađuje u web stranicu pomoću *HTML* oznake `script`, kojom se preglednika obavještava da sadržaj koji slijedi predstavlja programski kôd određenog tipa. Osim što je moguće direktno ugraditi kôd u stranicu, moguće je dati referencu na udaljeni sadržaj koji predstavlja odgovarajući kôd kojeg preglednik dobavlja u procesu učitavanja stranice. Primjer jednostavnog *JavaScript* programa u web pregledniku koji ispisuje poruku te je mijenja na pritisak gumba dan je u ispisu 2.3. Djelovanje *JavaScript* programa vidljivo je na slikama 2.3 i 2.4 koje prikazuju web stranicu u pregledniku prije i nakon stiska gumba, tj. izvršavanja *JavaScript* programa koji mijenja izgled i tekst stranice.

JavaScript engine je prevodioc koji interpretira *JavaScript* izvorni kôd i izvršava danu skriptu. Prvi takav prevodioc dio je *Netscape Navigator* Web preglednika i ostvaren je u C programskom jeziku, dok je noviji *Rhino engine* ostvaren u Java programskom jeziku.[7] Kako je *JavaScript* jedini programski jezik koji podržavaju svi moderni preglednici, postao je ciljni jezik raznih razvojnih okvira u drugim jezicima, iako izvorno nije bio tome namijenjen. Unatoč ograničenjima zbog dinamičke

Slika 2.3. Prikaz *HTML* dokumenta u Web pregledniku prije pritiska gumba



Slika 2.4. Prikaz *HTML* dokumenta u Web pregledniku nakon pritiska gumba

prirode jezika, sve veća brzina *JavaScript* virtualnih strojeva čine jezik smislenim ciljnim jezikom.

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
2    "http://www.w3.org/TR/html4/strict.dtd">
3  <html>
4    <head>
5      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
6      <title>Primjer programa</title>
7    </head>
8    <body onload="ispis('Dobar dan, svijete!', false);">
9      <div id="moj_tekst"></div>
10     <p>
11       <input type="submit" value="Pritisni me" onclick="ispis('
           Dovidenja, svijete!', true);"/>
12     </p>
13   </body>
14   <script type="text/javascript">
15       // ugradnja JavaScript koda u HTML stranicu
16       function ispis(vrijednost, bojati) {
17         moj_element = document.getElementById('moj_tekst');
18         moj_element.innerHTML = vrijednost;
19         if (bojati) {
20           moj_element.setAttribute('style', 'background-color:black;
                 color:red;');
21         }
22       }
23   </script>
24 </html>
```

Ispis 2.3. Primjer *JavaScript* programa u Web okolini

JavaScript se može ugraditi direktno u web stranicu, te se sadržaj dobavlja pomoću *HTTP* protokola metodom *GET*, kao što je opisano u poglavlju 2.4. Uz standardne objekte, klijentna *JavaScript* ima pristup posebnim objektima preglednika, tj. modelu *DOM*.

2.2.2. *JavaScript* objektna notacija

JavaScript objektna notacija (engl. JavaScript *Object Notation*) je jednostavan oblik za izmjenu računalnih podataka. Oblik je tekstualan i lako čitak ljudima, a podoban je za prikazivanje jednostavnih podatkovnih stuktura i asocijativnih polja,

tj. objekata. Najčešće je korišten za serijalizaciju i prijenos podataka preko mreže, a glavna uporaba mu je u programiranju Web primjenskih programa gdje služi kao alternativa *XML* obliku. Iako je *JSON* temeljen na podskupu *JavaScript* programskog jezika i obično se koristi s njim, smatra se jezično neovisnim podatkovnim oblikom.[5]

Korištenje *JSON* oblika u *AJAX* arhitekturi svodi se na to da poslužitelj šalje objekt serijaliziran u *JSON* obliku, a klijent taj zapis pretvara objekt odgovarajućim procesom i koristi ga dalje na klijentu. Uzmimo primjer *XML* zapisa podataka o jednoj datoteci danog u ispisu 2.4. Datoteka sadrži informacije o datumu i vremenu stvaranja, te korisnika koji ju je stvorio. Nadalje, datoteka sadrži dozvole za skupove korisnika koji ju smiju čitati i brisati. Istoznačna podatkovna struktura u *JSON* obliku dana je u ispisu 2.5.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <Datoteka ime="moja_datoteka">
3    <podaci>
4      <datum_stvaranja>12/7/1985</datum_stvaranja>
5      <vrijeme_stvaranja>01:00:00</vrijeme_stvaranja>
6      <korisnik>admin</korisnik>
7    </podaci>
8    <dozvola tip="citanje">*</dozvola>
9    <dozvola tip="brisanje">admin</dozvola>
10 </Datoteka>
```

Ispis 2.4. *XML* specifikacija datoteke

Deserijalizacijom objekta predstavljenog u ispisu 2.5 stvara se objekt koji predstavlja danu datoteku, kao što je prikazano u ispisu 2.6, pod pretpostavkom da varijabla `json_tekst` sadrži tekstualan serijaliziran objekt. Ono što čini *JSON* format vrlo korisnim i jednostavnim za uporabu je da su standardna sučelja za raščlanjivanje i stvaranje *JSON* kôda ostvarena u preko dvadeset programskih jezika kao što su *Java*, *JavaScript*, *C*, *C++*, *C#*, *Python* i *Ruby*, koji su danas jedni od najčešće korištenih programskih jezika. Korištenjem tih sučelja osigurava se točnost i dosljednost tumačenja zapisa i odstranjuju se najčešće greške i problemi sustava. Korištenje *JSON* oblika za prijenos podataka Web servisima umjesto *XML* oblika česta je tema debate zbog toga što *XML* format bolje predočava strukturu podataka kao dokument, dok *JSON* oblik prikazuje podatke bez previše troškova memorije na opis strukture. Iako se koriste oba oblika, potrebno je unaprijed odrediti i pretpostaviti kakve će podatke servis slati i primati, te prema tome odabrati onaj oblik

koji bolje odgovara određenim potrebama.

```
{
 "ime": "moja_datoteka",
 "podaci": {
     "datum_stvaranja": "12/7/1985",
     "vrijeme_stvaranja": "01:00:00",
     "korisnik": "admin",
 },
 "dozvola": [
     { "tip": "citanje", "vrijednost": "*" },
     { "tip": "brisanje", "vrijednost": "admin" }
 ]
}
```

Ispis 2.5. *JSON* specifikacija datoteke

```
// ili JSON.parse(json_text) u novijim preglednicima
var datoteka = eval("(" + json_tekst + ")");

print(datoteka.ime); // ispisuje "moja_datoteka"
print(datoteka.podaci.korisnik); // ispisuje "admin"
print(datoteka.dozvola[0].vrijednost); // ispisuje "*"
```

Ispis 2.6. Raščlanjivanje *JSON* zapisa

2.3. XML

XML (engl. *eXtensible Markup Language*) je jednostavan i fleksibilan format originalno oblikovan kako bi riješio probleme masivnog elektroničkog objavljivanja, no sve više i više igra i važnu ulogu u razmjeni raznih podataka na Internetu. Za razliku od *HTML* jezika, *XML* jezik služi isključivo za semantičku pohranu i prijenos podataka, dok se *HTML* koristi za prezentacijsku logiku tj. način na koji su podaci prikazani korisniku. *XML* je meta jezik za definiranje oznaka (engl. *tag*) i opisivanje strukturnih veza među njima.[2]

XML dokumenti se često koriste za spremanje konfiguracije primjenskih programa zato jer jednoznačni i manje skloni pogreškama, te su lako čitljivi korisnicima i računalima.

```
<?xml version="1.0" encoding="UTF-8" ?>
<korijen>
  <grana ime="lijeva">
    Lijeva grana
  </grana>
  <grana ime="desna">
    Desna grana
  </grana>
</korijen>
```

Ispis 2.7. Primjer *XML* dokumenta

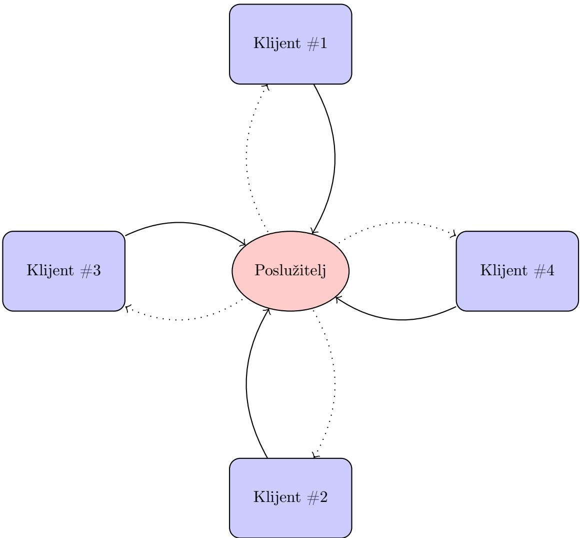
Dokumenti u *XML* formatu obvezno sadržavaju verziju i encoding dokumenta u prvoj nepraznoj liniji, a nakon toga ukorijenjeno stablo ugniježđenih tagova. Imena tagova su proizvolja, za razliku od oznaka u *HTML*. no ipak podliježu određenim pravilima kao što je ono da ime mora započeti slovom, ne smije početi sa podnizom *"xml"* i ne smije imati praznina. Svaki oznaka može sadržavati proizvoljan broj atributa i njihovih vrijednosti, te mora biti završena oznakom zatvaranja oznake kako bi dokument bio jednoznačan. Skupom oznaka i njihovih atributa moguće je definirati skup podataka i njihovu međusobnu ovisnost. U ispisu 2.7 prikazan je jednostavan *XML* dokument.

2.4. HTTP protokl

HyperText Transfer Protocol – HTTP je protokol aplikacijske razine za raspodijeljene informacijske sustave. *HTTP* je zasnovan na zahtjev-odgovor standardu prikazanom na slici 2.5, kao što je tipično za klijent-poslužitelj sustave. Klijent je primjenski program kao što je Web preglednik ili računalo koje koristi krajni korisnik, dok je poslužitelj tipično računalo na kojem se nalazi Web stranica. Klijent šalje *HTTP* zahtjev i prikazuje odgovor od posluživača na kojem su spremljeni ili dinamično stvoreni podaci koji se šalju klijentu, npr. *HTML* datoteke ili slike. Između klijenta i poslužitelja može biti više posrednika kao što su posrednički poslužitelji ili tuneli. *HTTP* nije ograničen na korištenje *TCP/IP* protokola za mrežnu komunikaciju, no takva ostvarenja su najpopularnija za Internet.

2.4.1. HTTP zahtjev

Da bi klijent zatražio neke podatke od poslužitelja, mora mu uputiti poruku u kojoj te podatke zahtijeva. *HTTP* protokol striktno definira što zahtjev mora



Slika 2.5. Zahtjev-odgovor standard

sadržavati. Prva linija u zahtjevu mora sadržavati metodu i traženu datoteku, te verziju protokola (npr. *HTTP/1.1*). Nakon toga, protokol zahtjeva rječnik zaglavlja, tj. skup uređenih parova imena atributa i njihovih vrijednosti koji definiraju razne karakteristike zahtjevane od traženih podataka. Nakon rječnika atributa, mora slijedini prazna linija, te neobavezna poruka.[6]

HTTP zahtjev podržava nekoliko metoda (engl. *verbs*) koji definiraju željenu akciju koju poslužitelj treba izvršiti nad zadanom datotekom. Hoće li datoteka biti već postojeći podatak na poslužitelju ili dinamično generiran sadržaj, ovisi o programskom ostvarenju web poslužitelja. Najvažnije metode *HTTP* zahtjeva su metoda *GET* i metoda *POST*. Metoda *GET* je sigurna metoda tj. ne mijenja stanje poslužitelja, a kojom se od poslužitelja zahtjeva prikaz određene datoteke, dok se metodom *POST* podaci predaju poslužitelju, tj. određenom sustavu na obradu. *HTTP* poslužitelji moraju ostvariti metode *GET* i *POST*, a uz njih se i najčešće ohrabruje ostvarenje metode *OPTIONS* koja za određeni *URL* daje popis *HTTP* metoda koje taj *URL* podržava.

```
GET /index.html HTTP/1.1
 Host: www.fer.hr
```

Ispis 2.8. Primjer *HTTP* zahtjeva

U ispisu 2.8 prikazan je jednostavan zahtjev za zadanu stranicu poslužitelja *index.html* koji dolazi sa *www.fer.hr* domene.

2.4.2. HTTP odgovor

Nakon što *HTTP* poslužitelj zaprimi zahtjev i obradi ga, klijentu natrag šalje odgovor. Prva linija odgovora se naziva linija stanja te sadrži kôd stanja i tekstualnu poruku koja ga opisuje, npr. *"404 Not Found"*. Način na koji klijent obrađuje odgovor primarno ovisi o kôdu stanja, a nakon toga o samom odgovoru, npr. samo ako je odgovor *"200 OK"* će klijent obrađivati sam odgovor dok će u slučaju pogreške prikazati korisniku obavijest o grešci ili dodatne informacije o tome što se dogodilo.

HTTP odgovor sadrži u prvoj liniji verziju protokola i stanje, a nakon toga, slično kao i u zahtjevu, skup parova atributa, tj. njihovih naziva i vrijednosti, svaki u svojoj liniji. Nakon toga dolazi prazna linija i onda sam sadržaj zahtjevane stranice u slučaju metode *GET*.[6]

```
HTTP/1.1 200 OK
Date: Tue, 17 Nov 2009 22:09:44 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Transfer-Encoding: chunked

<html>
  <head>
    <title>Odgovor</title>
  </head>
  <body>
    Ovo je jednostavan <em>HTTP</em> odgovor.
  </body>
<html>
```

Ispis 2.9. Primjer *HTTP* odgovora

HTTP odgovor u zaglavlju sadrži i tip sadržaja odgovora (engl. *content type*), te ovisno o tome različito tumači primljene podatke. U slučaju da je tip običan tekst (*text/plain*), sadržaj odgovora tumači se kao skup znakova, no ako je tip slika u *PNG* obliku (*image/png*), od klijenta se očekuje da shvati sadržaj poruke kao bajtovni zapis slike. Naravno, različiti klijenti mogu imati različita programska ostvarenja interpretacije podataka; dok će Web preglednik prikazati sliku korisniku na zaslonu, mrežni dohvatitelj (npr. *wget*)) će čitati odgovor kao bajtovni tok i spremati ga u lokalnu datoteku, ignorirajući semantiku podataka.

U ispisu 2.9 prikazan je jednostavan *HTTP* odgovor koji sadrži stanje odgovora (*"200 OK"*), skup konfiguracijskih *HTTP* zaglavlja te sam sadržaj stranice u *HTML* obliku.

2.5. AJAX

Asinkronai *JavaScript* i *XML* (engl. *Asynchronous JavaScript and XML – AJAX*) je skup međusobno povezanih tehnika razvoja Internet primjenskih programa na strani klijenta u svrhu stvaranja dinamičnih i interaktivnih sustava. Koristeći *AJAX*, klijentski primjenski program može asinkrono, pozadinski dobaviti podatke sa poslužitelja ne ometajući time ponašanje i doživljaj stranice. Podaci se najčešće dobavljaju korištenjem *XmlHttpRequest* objekta ugrađenog u sve modernije preglednike.[10]

XmlHttpRequest(XHR) je sučelje za primjensko programiranje *DOM* modela koje omogućava da se preglednikovim skriptnim jezicima kao što je *JavaScript* šalje *HTTP* ili *HTTPS* zahtjev direktno na poslužitelj i prima odgovor poslužitelja direktno u skriptni jezik. Kad podaci pristignu na klijent, može ih se čitati kao običan tekst, ili pak kao *XML* strukturu u slučaju da su u *XML* obliku.

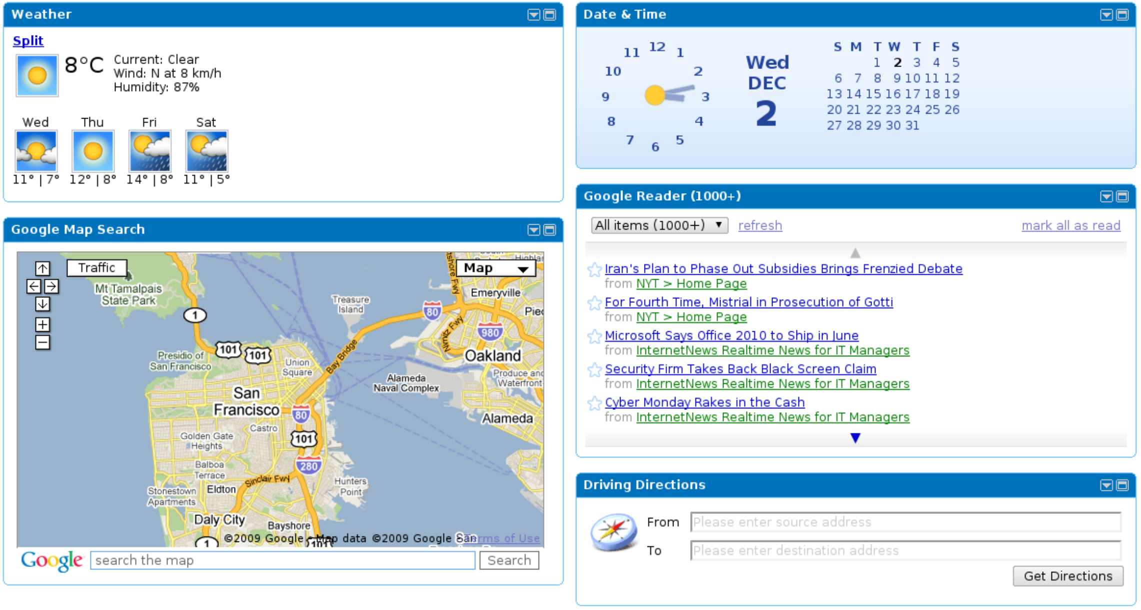
Kako bi se pripremio *HTTP* zahtjev koji će se slati na poslužitelj, koristi se `open` metoda *XHR* objekta kojoj se zada željena *HTTP* metoda i adresa poslužitelja kako bi se zahtjev stvorio. Prema W3 standardima, ne bi se trebale dopuštati adrese koje su na drugim poslužiteljima radi sigurnosti. Metodom `setRequestHeader` se mogu postaviti zaglavlja *HTTP* zahtjeva, dok se metodom `send` šalje zadani sadržaj na poslužitelj. Kada preglednik zapremi asinkroni odgovor, poziva funkciju `onreadystatechange` u kojoj se očekuje logika za obradu odgovora.[13]

Najbolji primjeri korištenja *AJAX* tehnologija su Google Maps, Gmail i Google Wave kao primjenski programi koji sadržavaju dinamično sučelje i njihova komunikacija sa poslužiteljima ne blokira klijentski prikaz za vrijeme dohvata.

2.6. Google Gadget sustav

Google udomljenici su dinamičan Web sadržaj koji se može smjestiti na bilo koju stranicu na Internetu. Udomljenici mogu biti igre, mali primjenski programi, geografske karte ili bilo koji drugi sadržaj. Udomljenici se mogu ugraditi na iGoogle stranicu kao što je prikazano slikom 2.6, ali i na bilo koju privatnu stranicu koja

ima pristup Google poslužiteljima.



Slika 2.6. Prikaz više udomljenika u *iGoogle* okolini

Google udomljenik se sastoji od nekoliko jednostavnih građevnih jedinica: *XML*, *HTML* i *JavaScript*.

XML je predodžbeni jezik opće svrhe koji služi za opisivanje strukturiranih podataka na način da ih i ljudi i računala mogu jednostavno čitati i pisati. Specifikacija udomljenika se piše u *XML* jeziku, tj. udomljenik je samo *XML* datoteka smještena negdje na Internetu gdje ju Google može pronaći. Specifikacija sadrži niz instrukcija kako obraditi i prikazati udomljenik, a može sadržavati sve potrebne podatke u sebi ili reference na vanjske resurse u *URL* obliku.[9].

HTML je predodžbeni jezik koji služi za oblikovanje Internet stranica. Statični sadržaj udomljenika je obično napisan u *HTML* obliku. *HTML* jezik nalikuje *XML* obliku, ali se ne koristi za opis strukturiranih podataka već za semantički prikaz stranica.

JavaScript je skritpni programski jezik koji se koristi za dodavanje dinamičnog ponašanja u udomljenik omogućavajući međudjelovanje korisnika sa sadržajem udomljenika.

2.6.1. XML struktura

Kao što je vidljivo u ispisu 2.10, *XML* specifikacija udomljenika sastoji se od tri važna dijela: sadržaja, korisničkih postavki te postavki udomljenika.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
  <ModulePrefs title="Primjer udomljenika"
  author="Goran Petrovic" height="250" />
  <UserPref name="moja_postavka"
        display_name="Postavka 1"
        datatype="enum"
        default_value="4">
    <EnumValue value="3" display_value="Odabir 1"/>
    <EnumValue value="4" display_value="Odabir 2"/>
    <EnumValue value="5" display_value="Odabir 3"/>
  </UserPref>
  <Content type="html">
  <![CDATA[
      <div style="text-align:center">
          Ovo je primjer Google udomljenika.
          <a id="moj_link" title="Moj udomljenik" target="_blank"
          href="moj_url">
          <img border="1" alt="Slika" src="moja_slika.jpg"
          title="Moj gumb."></a>
      </div>
  ]]>
  </Content>
</Module>
```

Ispis 2.10. *XML* specifikacija Google udomljenika

Sadržaj (engl. *Content* – *<Content>*) je mjesto gdje se definira stvarna programska logika udomljenika a često i *HTML* elementi i njihov izgled. Tip sadržaja može bili *html*, ako se sadržaj nalazi u udomljeniku, ili *url* ako je sadržaj na udaljenom poslužitelju.

Korisničke postavke (engl. *User Preferences* – *<UserPrefs>*) su odjeljak koji definira kontrole za postavljanje postavki udomljenika. Na primjer, udomljenik može sadržavati polje za upis imena korisnika koje će udomljenik pamtiti i koristiti u budućnosti. Postavke mogu biti raznih podatkovnih tipova kao što su liste, nizovi, polja, brojevi itd.

Slika 2.7. Prikaz udomljenika u Web orkužju

Postavke udomljenika (engl. *Gadget Preferences* – *<ModulePrefs>*) su odjeljak koji određuje karakteristike udomljenika kao što su naslov, autor, veličina itd. Prema ovim postavkama, na Google pretraživaču udomljenika će se prikazivati slika, autor i ključne riječi.

Izgled udomljenika danog na ispisu 2.10 u *iGoogle* razvojnoj okolini prikazan je na slici 2.7.

2.6.2. *OpenSocial* okruženje

OpenSocial je skup API sučelja za mrežne društvene primjenske programe. Svi primjenski programi koji ostvaruju *OpenSocial* sučelja mogu međudjelovati.[16]

Sustav je temeljen na *HTML* i *JavaScript* tehnologijama i na sustavu Google udomljenika i omogućava komuniciranje sa poslužiteljem koji daje pristup korisnikovim kontaktima i njihovim informacijama.

Poseban imenski prostor `gadgets.io` omogućava dobavljanje udaljenog sadržaja unutar udomljenika bez direktnog korištenja *XmlHttpRequest* objekta jednostavnim pozivom funkcije `gadgets.io.makeRequest` koja uzima za parametre *URL* adresu udaljenog sadržaja i funkciju koja obrađuje odgovor udaljenog poslužitelja. Funkcija koja obrađuje odgovor kao parametar uzima objekt koji sadrži dohvaćeni udaljeni sadržaj, bio on u tekstualnom obliku, zapis slike ili *PDF* dokument.

2.7. Google App Engine sustav

Google App Engine je platforma za razvijanje i udomljavanje Web primjenskih programa u Google podatkovnim centrima, a temelji se na raspodijeljenom sustavu – računalnom oblaku koji virtualizira primjenske programe preko više poslužitelja i podatkovnih centara. *Google App Engine* sustav dopušta korisnicima da pokreću svoje Web primjenske programe na Google infrastrukturi, ne brinući se o poslužiteljima, bazama podataka, upravljanju prometa ili zagušivanju.

Google App Engine podržava primjenske programe pisane u više tehnologija. Sa *App Engine Java runtime* okružjem, moguće je pokretati primjenski program koristeci standardne *Java* tehnologije, uključujući *JVM, Java Servlets* i *Java* programski jezik, kao i bilo koji drugi jezik koji koristi programski prevodioc baziran na *JVM* kao sto su *JavaScript*, *Ruby* ili *Scala*. *App Engine* nativno podržava i *Python runtime* okužje koje uključuje brzi interpreter i *Python* standardne biblioteke. Oba okružja su namijenjena za brzo i sigurno izvršavanje primjenskog programa neovisno o drugim programima na sustavu.

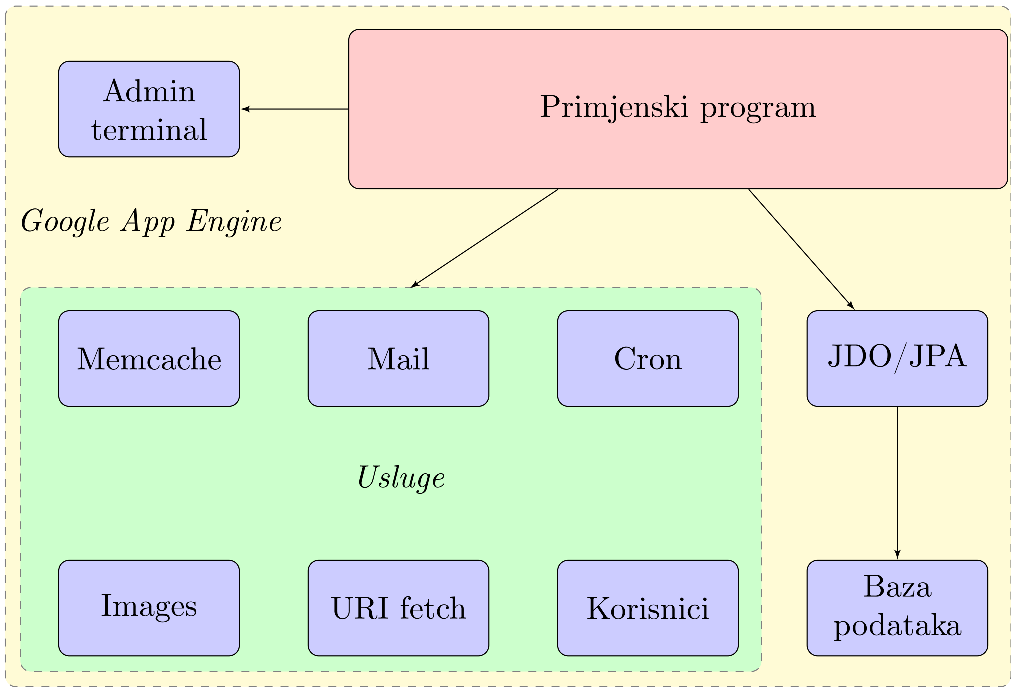
App Engine sadrži rješenja za dinamično posluživanje stranica, trajno pohranjivanje podataka sa upitima i transakcijama, te automatsko balansiranje prometa. Dodatno, interno rješava autentikaciju korisnika preko *Google Accounts API*.

2.7.1. Java Runtime okruženje

Java Runtime okruženje koristi *Java 6* verziju Jave te sadrži *Java SE Runtime Environment (JRE) 6* platformu i biblioteke. Ograničenja okruženja su ostvarena u samom Java virtualnom stroju (engl. *Java Virtual Machine – JVM*), tj. pokušaj korištenja nedopuštenih značajki kao što je otvaranje mrežnog priključka ili datoteke će rezultirati iznimkom za vrijeme izvođenja, na taj način osiguravajući da se ne naruši sigurnost programa.

Trajno pohranjivanje podataka moguće je korištenjem *App Engine Datastore* centralne baze podataka pomoću *Java Data Objects (JDO)* i *Java Persistance API (JPA)* sučelja. Podaci se spremaju u *BigTable* bazu podataka koja je oblikovana tako da sprema petabajte podataka na rasprostranjeni sustav od preko stotinu tisuća računala.

Sustav za slanje elektroničke pošte dostupan je u *App Engine* sustavu pomoću *JavaMail API* sučelja, dok je dohvat *URL* adresa moguć korištenjem `java.net`



Slika 2.8. Arhitektura Google App Engine sustava za Javu

HTTP API funkcionalnosti.[4]

App Engine dodatno omogućava *memcache* – programsko ostvarenje raspodijeljenog podatkovnog skladišta u memoriji koji se interno brine o topologiji sustava na kojem se izvršava primjenski program. Korištenje raspodijeljenog podatkovnog skladišta znatno pospješuje brzinu podatkovno opterećenih sustava bez ikakvih postavki.[1]

Na slici 2.8 prikazana je arhitektura *Google App Engine* sustava za programski jezik Java.

2.7.2. Java Servlet tehnologija

Java Servlet tehnologija pruža jednostavan i dosljedan mehanizam za proširenje funkcionalnosti Web poslužitelja i pristupanje postojećim sustavima. Servleti omogućuju izgradnju platformski neovisnih Web primjenskih programa bez ograničenja *CGI* programa. Servleti imaju pristup svim *Java API* sučeljima, uključujuči *JDBC API* za povezivanje sa bazama podataka. Oni također imaju prisup *HTTP* specifičnim bibliotekama te koriste sve privilegije zrelog Java jezika, uključujući prenosivost, performanse i ponovnu iskoristivost. Korištenje *JavaServer Pages* (*JSP*) tehnologije u korak sa servletima omogućava jednostavno dinamično posluži-

vanje *HTML* i *XML* sadržaja, odvajajući prezentacijsku logiku od poslovne. Servlet tehnologija jednostavno je proširiva i lako prilagodljiva novim sustavima.[11]

Korištenjem Servleta elimira se mnogo sigurnosnih problema koji nastaju izvršavanjem naredbi operacijskog sustava kao što su čitanje elektronske pošte ili brisanje datoteka. Java jezik omogućava večinu potrebne funkcionalnosti i time ne odstranjuje mogućnost izvršavanja opasnih naredbi.

2.7.3. Java Data Objects

Java Data Object (JDO) je specifikacija trajnog pohranjivanja Java objekata bez potrebe ostvarenja određenih sučelja ili nasljeđivanja posebnih razreda. *JDO* su obične instance razreda programskog jezika Java te su kao takvi transparentni prema poslužiteljima trajnog pohranjivanja. *JDO* definira sučelja za anotiranje Java objekata, dobavljanje objekata pomoću upita te međudjelovanje sa bazom koristeći transakcije.

Primjenski program koji koristi *JDO* sučelje može raditi sa različitim bazama podataka bez korištenja ikakvog kôda svojstvenog poslužitelju baze, čineći tako korištenje relacijskih, hijerarhijskih i objektnih baza podataka i njihovu izmjenu jednostavnima.

App Engine pruža opciju tzv. poboljšavanja razreda, tj. korak procesiranja razreda nakon prevođenja gdje se prema naznačenim anotacijama određuje na koji način će se razred obnoviti iz trajne memorije u radnu.

3. Udomljenik za izradu korisničkog sučelja

Udomljenik za izradu korisničkog sučelja služi za jednostavno uređivanje grafičkih elemenata za druge udomljenike.

U poglavlju 3.1 opisana je uloga udomljenika tijekom izgradnje potrošačkih primjenskih programa. Primjena *Toolbox* udomljenika u razvoju drugih udomljenika i način korištenja iznešeni su u poglavlju 3.2.

3.1. Uloga i funkcija oblikovanog udomljenika

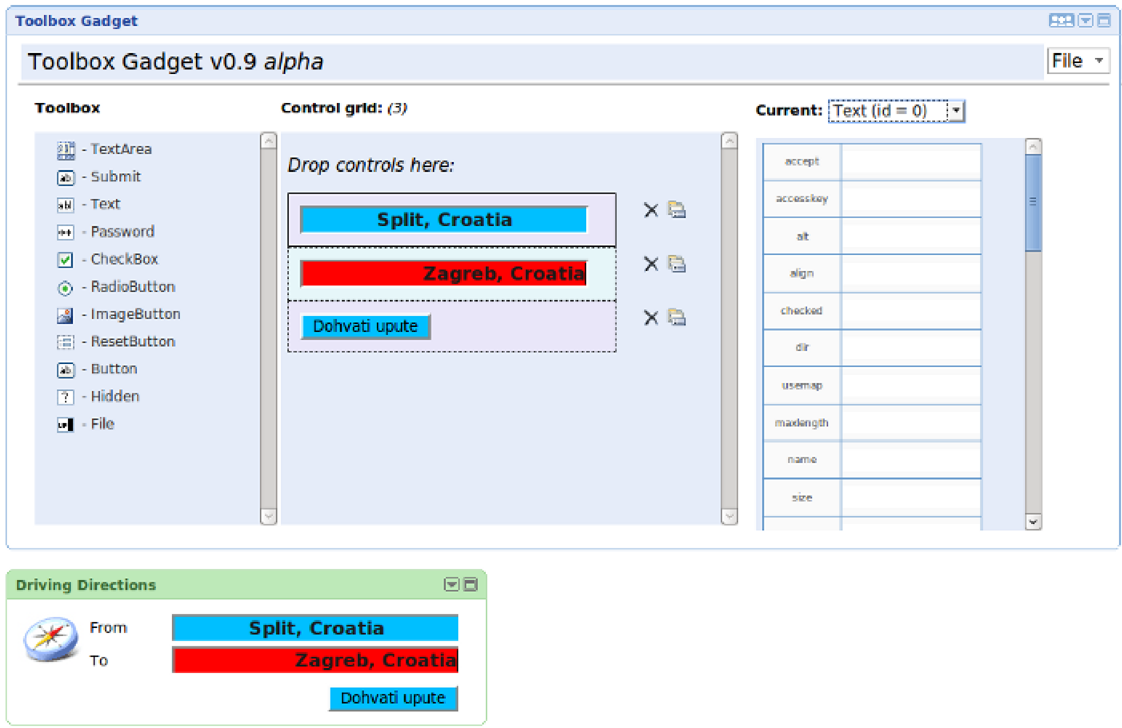
Udomljenik za izradu korisničkog sučelja je jednostavan primjenski program kojemu je svhra stvaranje i uređivanje *HTML* elemenata direktno u *web* pregledniku za naknadno korištenje pri izradi drugih udomljenika. Funkcija mu je biti lako dostupnim alatom ne odvraćajući pozornost s projekta za koji se koristi.

Stvaranje *HTML* elemenata vrši se *drag-and-drop* paradigmom, tj. odabrani se element odvuče na radnu površinu gdje ga je poslje moguće uređivati i prilagoditi potrebama putem intuitivnog korisničkog sučelja. Tematski generirani elementi se jednostavnim postupkom koriste više puta, tako eliminirajući redundanciju.

3.1.1. Funkcija udomljenika u razvoju drugih udomljenika

Mogućnosti modernih *web* preglednika daleko nadilaze funkcionalnost onih u začetku Interneta, a prihvaćanjem *HTML 5* standarda samo će ubrzati napredak do te mjere da će *web* preglednik koristiti 3D mogućnosti grafičke kartice i pokretati pozadinske dretve, tako odrađujući posao ne zaustavljajući korisničko sučelje. Približavanje izrade *web* primjenskih programa krajnjem korisniku sve je važnije, ali zbog rasta funkcionalnosti i teže.

Pri izradi skupa alata, potrebno je korisniku omogućiti što jednostavnije privikavanje na standardizirano sučelje koje treba koristiti iste boje, oblike i metode razmjene podataka sa korisnikom, a da bi se ostvarenje tog principa olakšalo, nužno je izgraditi standardan skup elemenata kao što su gumbi, elemenati za upis teksta i padajući izbornici, koji će svi nositi oznaku istog projekta.



Slika 3.1. Uporabe Toolbox udomljenika u izradi drugih udomljenika

Izrada udomljenika u *iGoogle* okružju uključuje opisivanje *HTML* elemenata, uređivanje istih pomoću *CSS* (engl. *Cascading StyleSheets*) tehnologije i povezivanje sadržaja pomoću *JavaScript* programskog jezika. Udomljenik za izradu korisničkog sučelja sudjeluje u tom procesu u dijelu opisivanja *HTML* elemenata tako da nadohvat ruke nudi *in situ* uređivanje i repliciranje potrebnih elemenata. Korištenjem udomljenika, ideja je stvoriti i urediti sve za projekt potrebne *HTML* elemente te ih spremiti u repozitorij. Pri oblikovanju novih udomljenika iz iste obitelji, koriste se uvijek isti elementi koji, jednom spremljeni, stoje na dohvat klika mišom te se brzo i jednostavno ugrađuju u novi sadržaj.

Na slici 3.1 prikazan je Toolbox udomljenik sa tri uređena tri elementa: dva polja za unos teksta i jedan gumb, koji iskorišteni za uređivanje udomljenika za dohvat uputa za putovanja. Osim što Toolbox udomljenik odvaja funkcionalnu logiku i oblikovanje primjenskih programa, smanjuje redundante korake i moguće

greške u oblikovanju osiguravajući korištenje uvijek istog stila.

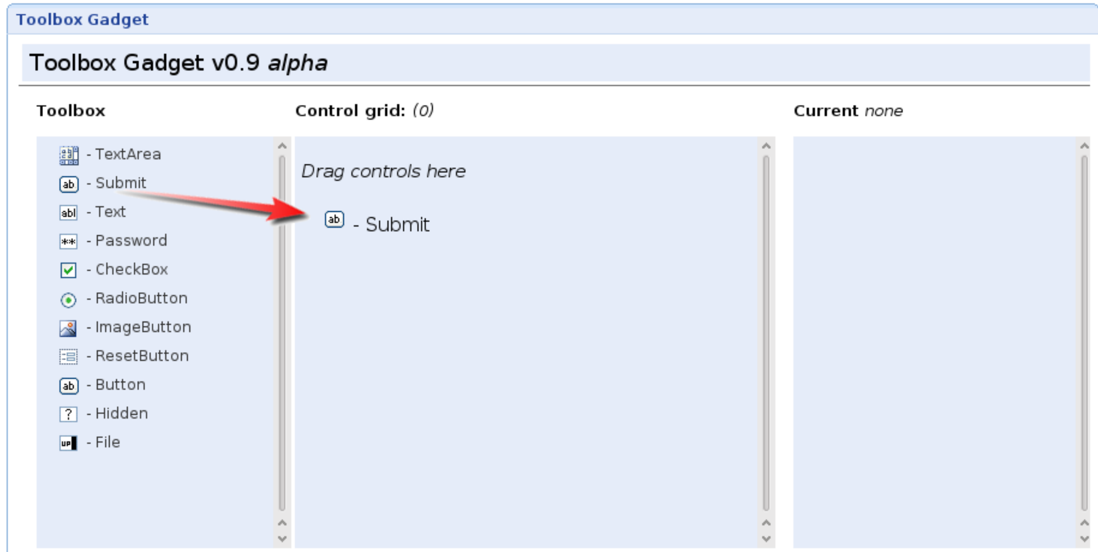
3.2. Primjena udomljenika

Osnovna smisao Toolbox udomljenika je uređivanje pojedinih *HTML* elemenata. Kako bi elementi uredili, potrebno ih je dodati na radnu površinu, stvarajući tim jedinku tog tipa, tj. pravi element spremljen u model primjenskog programa koji se iscrtava na radnoj površini ovisno o postavkama koje uređuje korisnik.

Element na radnoj površini mijenja izgled i ponašanje prema unesenim postavkama tako da se njegov zapis u modelu osvježi novim podacima. Kako je korisničko sučelje udomljenika građeno od nekoliko prikaza paradigme model-prikaz-upravljač, čim se dogodi neka promjena u modelu, svi prikazi se osvježe.

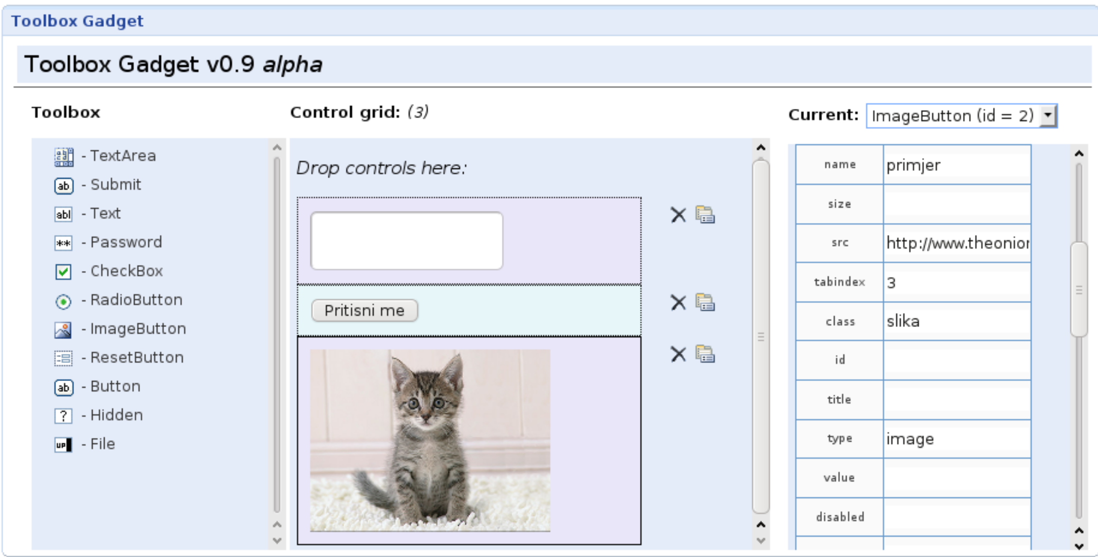
3.2.1. Radna površina

Elementi se iz alatne trake dodaju na radnu površinu metodom vuci i ispusti (engl. *drag-and-drop*) kao što je prikazano na slici 3.2.



Slika 3.2. Dodavanje elemenata na radnu površinu

Ispuštanjem elementa, u model se dodaje zapis za taj element sa svim atributima na zadanim početnim vrijednostima. *HTML* element se prikazuje u pravokutniku na radnoj površini, te su uz njega prikazani i gumbi za brisanje elementa i uređivanje *CSS* atributa elementa.



Slika 3.3. Uređivanje osnovnih atributa

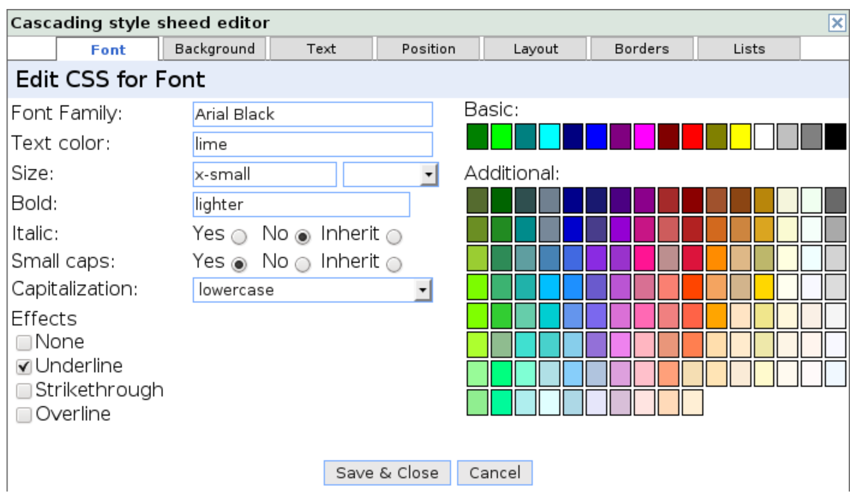
Brisanjem elementa, on se briše iz modela, a radna površina se osvježi, ne prikazujući više izbrisani element. Pritiskom na gumb za uređivanje izgleda elementa, otvara se posebno logički podijeljeno sučelje za uređivanje izgleda elementa. U slučaju da se na radnoj površini nalazi više elemenata, željeni se može označiti pritiskom na pravokutik u kojem se nalazi ili odabirom iz padajućeg izbornika trenutnog elementa (engl. *current*). Odabrani element ima crni okvir oko pripadajućeg pravokutnika, i njegove postavke se mogu uređivati u traci za postavke desno.

3.2.2. Uređivanje atributa HTML elemenata

Atributi *HTML* elementa dijele se na dva semantički odvojena dijela: uređenje izgleda elementa postavljanjem `style` atributa i uređenje ostalih *HTML* atributa.

Uređenje izgleda elementa vrši se kombiniranjem više *CSS* pravila, a kako kompliciranije oblikovanje iziskuje velik broj pravila i poznavanje točne sintakse, pravila ugnijež đivanja i skup mogućih vrijednosti kao što su heksadecimalni zapisi boja, atribut `style` se uređuje pomoću posebnog sučelja za oblikovanje izgleda elementa. Ostali atributi odabranog elementa jednostavno se uređuju promjenom u traci postavki desno. Ako je za atribut određen regularni izraz, unesena vrijednost mora biti zadanog oblika da bi se mogla postaviti.

Kao što je vidljivo na slici 3.3, postavljanjem vrijednosti atributa u traci za postavke, element se automatski ažurira. Postavljeni atribut izvora slike (engl. *source – src*) određuje koja se slika prikazuje, dok ime elementa (engl. *name*) ime-



Slika 3.4. Uređivanje izgleda elementa

nuje element tako da se u budućnosti uvijek može dohvatiti *DOM* funkcionalnošću preglednika.

3.2.3. Uređivanje style atributa

Uređenje izgleda elementa složen je proces te je stoga izvedeno pomoću posebnog sučelja u kojem su logički grupirana svojstva izgleda i lako su urediva.

Osnovni izgled elementa može se odvojiti na oblik, boju i stil slova (engl. *font*), postavke pozadine (engl. *background*), smjer teksta i udaljenost slova (engl. *text*), smještaj i veličina elementa (engl. *position*), raspored (engl. *layout*), postavke granica elementa (engl. *borders*) i postavke za liste (engl. *lists*), kao što je vidljivo na slici 3.4. Za svaki *CSS* atribut, ponuđen je izbor mogućih vrijednosti kao što su najčešće boje za tekst ili pozadinu, stupanj podebljanosti slova ili horizontalno i vertikalno poravnanje unutar elementa.[15]

Zatvaranjem sučelja za uređenje izgleda Toolbox udomljenika, proračunava se nova vrijednost `style` atributa tako da se raščlanjuju postavke iz sučelja i on njih gradi skup parova ključ-vrijednost za *CSS* atribute, te se dobivena vrijednost postavlja kao vrijednost `style` atributa uređivanog *HTML* elementa. Element automatski biva osvježen kako bi odražavao nove postavke. Unos atributa za izgled udomljenika većinom je automatski stvorena kombinacijom ponuđenih izbora u sučelju za uređivanje udomljenika, no kako nije izvodljivo ponuditi skup svih mogućih vrijednosti za atribute kao što su boja ili podebljanost slova, provjera smislenosti

korisnički unesenih vrijednosti ostaje na samom korisniku. Trenutno stajne na radnoj površini moguće je spremiti na poslužitelj za naknadno korištenje pomoću izbornika za upravljanje kontekstom. Jednom spremljeno stanje uvijek se može rekonstruirati.

4. Arhitektura udomljenika za oblikovanje korisničkog sučelja

Udomljenik se sastoji od dvaju logički odvojenih djelova: klijenta i poslužitelja (slika 4.3). Klijentu je zadatk prikazati korisničko sučelje i omogućiti korisniku dodavanje i uređivanje *HTML* elemenata, dok poslužitelj služi kao spremište konteksta: skup uređenih elemenata može se spremiti na poslužitelju pod proizvoljnim imenom za kasnije korištenje. Poslužitelj omogućuje spremanje, dohvaćanje i brisanje konteksta specifičnog korisniku: svaki korisnik ima svoj identifikator i vidi samo svoj repozitorij.

4.1. Arhitektura korisničkog sučelja

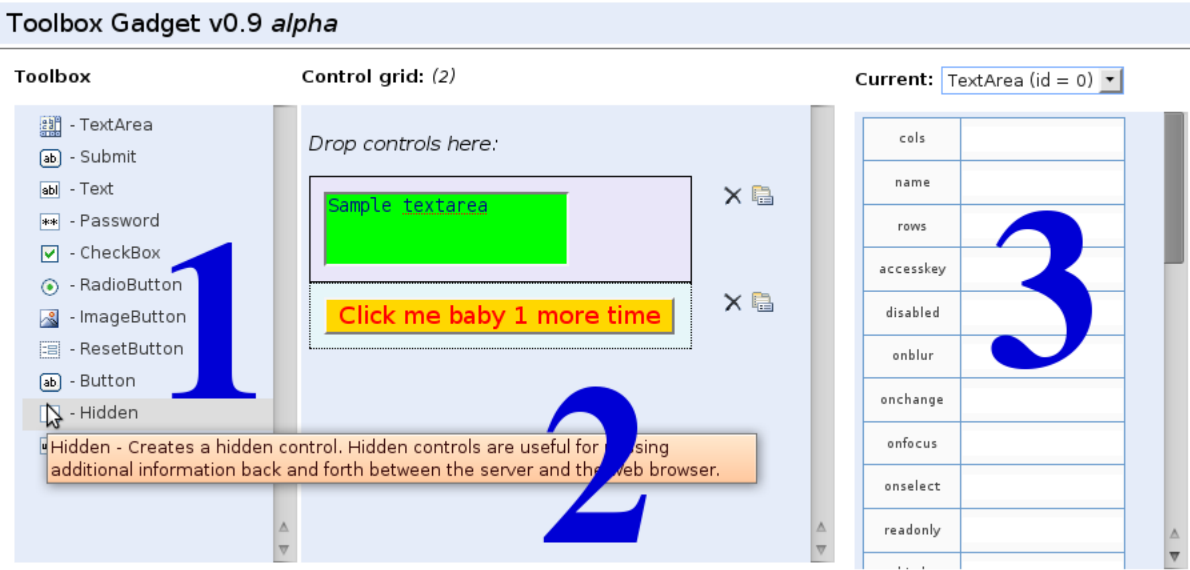
Kao što je prikazano na slici 4.1, udomljenik se sastoji od tri osnovna dijela, i to alatne kutije, radne površine i odjeljka za uređivanje svojstava.

Alatna kutija se sastoji od niza HTML elemenata koji se mogu dodati na radnu površnu. Lista elemenata se dinamički učitava iz specifikacije u `XML` obliku. Specifikacija sadrži metapodatke za element kao što su niz atributa, njihovi opisi i regularni izrazi za validaciju vrijednosti i lako je urediva, čineći tako dodavanje novih elemenata vrlo brzim i jednostavnim.

Radna površina sadrži sve uređivane elemente i omogućuje odabir trenutnog elementa koju se svojstva mogu postaviti u traci svojstava nadesno. Uz sam element, na radnoj površini se nalaze dva kontrolna gumba - jedan za brisanje elementa a drugi za uređivanje *CSS style* atributa. Uređivanje *CSS* stila je podjeljeno u više logičkih skupina i proteže se na nekoliko odjeljaka o što je više opisano u poglavlju 3.2.3.

Popis svojstava predstavlja sučelje za uređivanje svojstva predefiniranih u *XML*

specifikaciji. Svojstva se mogu mijenjati dok god nova vrijednost odgovara regularnom izrazu za to svojstvo, npr. `tabindex` mora biti pozivitan prirodan broj. Nakon promjene svojstva, model se osvježi s novim podacima, a samim time se promjeni i prikaz na radnoj površini, ako je promijenjeno svojstvo koje utječe na izgled elementa.



Slika 4.1. Osnovi djelovi udomljenika

Korisničko sučelje udomljenika sadrži više elemenata koji prikazuju iste podatke na različit način, kao što su radna površina, odjeljak za uređivanje svojstava i tekstualna lista elemenata koji se nalaze na radnoj površini. Ostvarenjem svakog elementa korisničkog sučelja zasebno stvara nepotrebne viškove u komunikaciji svakog elementa sa zajedničkim modelom, a kompleksnot ostvarivanja novih elemenata korisničkog sučelja koji daju drugačiji pogled na iste podatke postaje raste sa brojem elemenata.

Primjenom paradigme model-prikaz-upravljač na udomljenik, svi korisni podaci se grupiraju u model koji je ostvaren na način da osim što sadrži sve podatke, omogućava obavljanje svih potrebnih radnji nad podacima. Za svaki element sučelja oblikovan je odgovarajući prikaz, kako nalaže paradigma, koji sadrži isključivo prezentaciju logiku i metode za obavještavanje sustava o akcijama korisnika. Na slici 4.1 pod brojem 1 prikazana je radna površina – najvažniji element korisničkog sučelja koji prikazuje elemente koji se uređuju. Element sadrži skup *HTML* elemenata koji se dinamično dodaju i brišu sa površine, prateći odgovarajuće korisnikove akcije. Brojem 2 naznačena je mreža svojstava odabranog *HTML* elementa. Svoj-

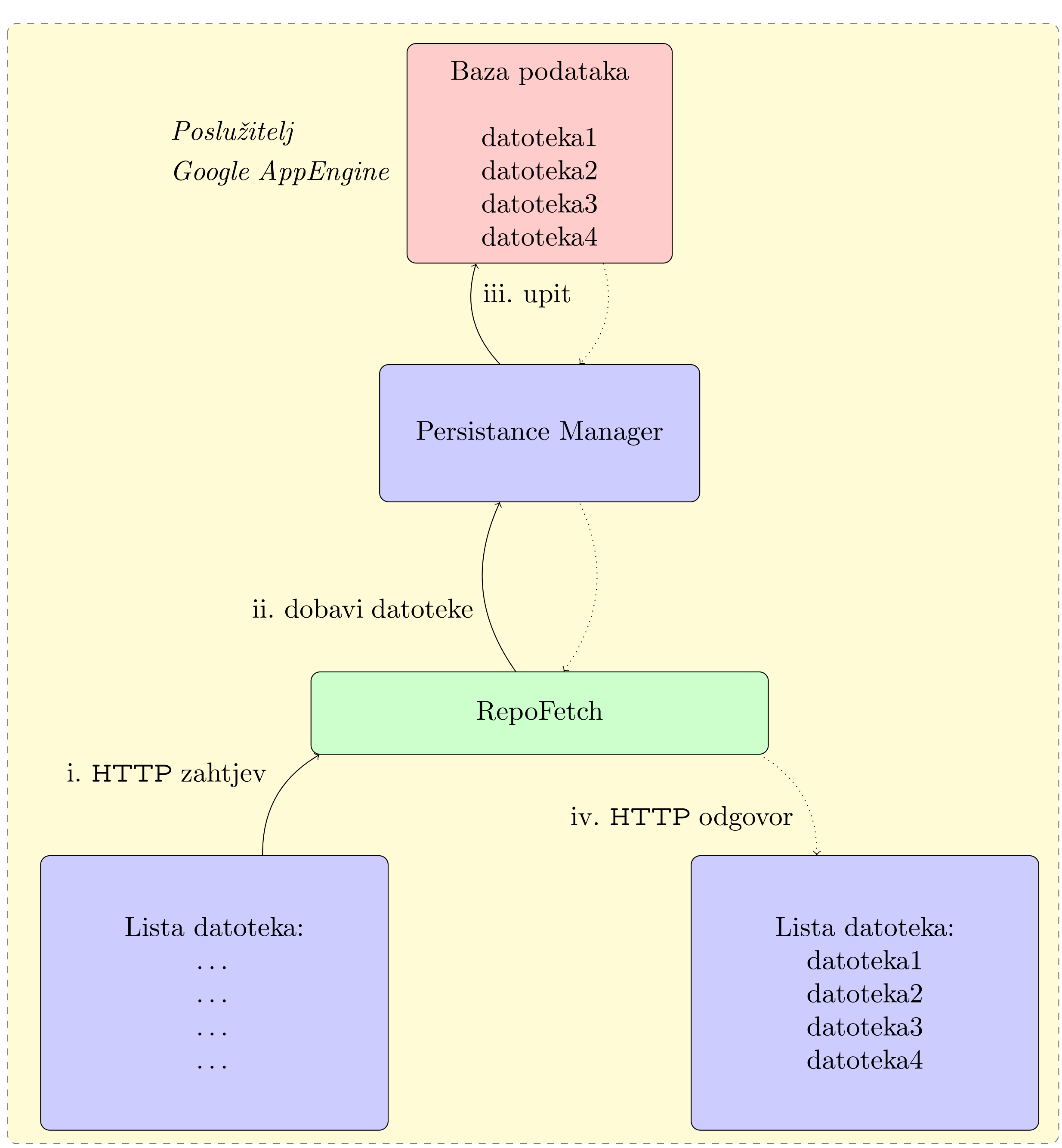
stva su opisana u tablici u kojoj je moguće unositi i mijenjati vrijednosti. Povišе mreže svojstava elemenata nalazi se padajući izbornik u kojem se postavlja odabrani element. Imajući u vidu da svi ovi elementi dijele isti model, kao poveznica između modela i prikaza po nalogu paradigme ostvaren je upravljač koji reagira na akcije korisnika i obaviještava prikaze o promjenama.

4.2. Arhitektura repozitorija na poslužitelju

Repozitorij na poslužitelju služi za spremanje i obnovu konteksta udomljenika i sagrađen je na *Google App Engine* arhitekturi koji podatke sprema pomoću *Persistance managera*. Za svakog korisnika, datoteke se spremaju u *App Engine Datastore*, i to koristeći Java podatkovne objekte (engl. *The Java Data Objects*), tj. serijalizirajući objekte koji sadrže podatke u bazu podateka. Instance serijaliziranih razreda mogu se koristeći istu tehnologiju rekonstruirati, te koristiti za obradu korisnikova zahtjeva. Zahtjev korisnika sadrži ključ tog korisnika kojim se identificira o kojem je korisniku riječ, tj. pri dobavljanju podatkovnih objekata iz memorije, u upit se dodaje ključ kako bi svaki korisnik imao odvojen repozitorij. Repozitorij je izveden tako da osluškuje zahtjeve od udomljenika i odgovori traženom informacijom.

Slika 4.2 prikazuje slijed komunikacije pri dohvatu popisa raspoloživih datoteka na repozitoriju. U prvom koraku, klijent zahtjeva popis datoteka (i). U drugom koraku, poslužitelj traži popis svih datoteka od sustava za trajno pohranjivanje podataka (ii). U trećem, prozirnom koraku, sustav za trajno pohranjivanje podataka šalje upit za traženim podacima trajnom spremištu kao što je baza podataka (iii). U četvrtom koraku, spremište podataka odgovara sustavu za trajno pohranjivanje, koji taj odgovor vraća primjenskoj logici na poslužitelju te konačno klijentu (iii), čime završava ciklus dohvata popisa raspoloživih datoteka.

Za potpunu funkcionalnost repoiztorij mora odgovarati na više vrsta zahtijeva, i to na zahtjev za popis raspoloživih datoteka, zahtjev za isporukom određene datoteke, zahtjev za spremanje datoteke i zahtjev za brisanje određene datoteke. Funkcionalnošću *HTTP servlet* tehnologije, repozitorij prima zahtjev i ovisno o zahtjevu, izvršava odgovarajuće radnje, te obavještava udomljenik o rezultatu. Udomljenik na poruku reagira obavještavanjem korisnika o daljnjem razvoju ili prikazivanjem određenog odbavljenog sadržaja.



Slika 4.2. Arhitektura repozitorija

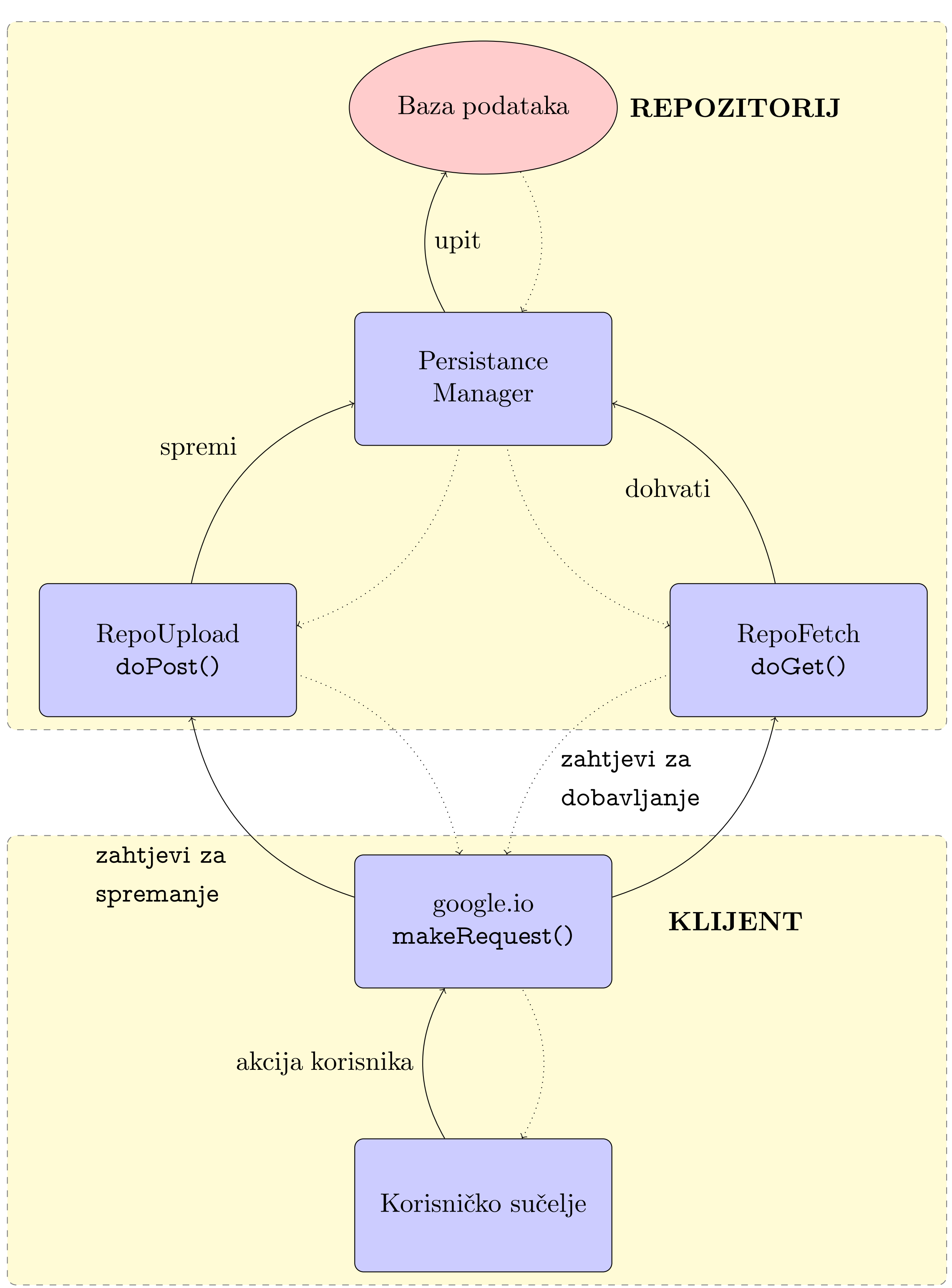
4.3. Povezivanje klijenta i poslužitelja

Povezivanje klijenta i poslužitelja česta je zadaća modernih Web primjenski program i najčešće se postiže radom u predefiniranim okolinama koje znaju međusobno komunicirati i dopuštaju niz metoda kojim se izmjena podataka može vršiti, ne brinući se o ostvarenju komunikacijskog sloja korištenog sustava. Neki primjeri takvog pristupa su Web servisi koji generirajući *proxy* razrede za klijent omogućavaju jednostavnu transparentnost podataka i izvršavanje poziva, i primanje podataka u željenom formatu ili baze podataka za koje postoje mehanizmi (npr. *JDBC - Java Database Connectivity API*) koji omogućavaju jedostavno sučelje za komunikaciju sa poslužiteljem, ne brinući se o implementaciji. U slučaju Toolbox Gadgeta, korištenje komunikacijskog sustava više razine nije moguće zbog ograničenja samog okruženja, te je komunikacija ostvarena na razini `HTTP` protokola.

Google Gadget okolina ne dopušta korištenje *XmlHttpRequest* objekta za asinkrone zahtjeve prema Web poslužitelju opisanog u poglavlju 2.5 radi sigurnosti, u ovom slučaju Java *servletu* koji osluškuje upite za repozitorij, ali pruža tu mogućnost pomoću ugrađenog `gadgets.io` imenskog prostora koja omogućuje upućivanje `GET` i `POST` metoda.[18] Klijent i poslužitelj u udomljeniku za izradu korisničkog sučelja povezani su preko tog sučelja opisanog na slici 4.3 i to u slučaju da korisnik zatraži podatke od poslužitelja ili kada mu ih šalje.

Kada korisnik zatraži popis datoteka u repozitoriju, upućuje se `HTTP GET` metoda poslužitelju *RepoFetch* s trenutnom korisničkom oznakom. Poslužitelj poslužuje zahtjev tako da koristeči korisničku oznaku za autentikaciju, zatraži od *Google App Engine* sustava podatke spremljene za danog korisnika te generira listu spremljenih datoteka. Poslužitelj odgovara korisniku sa *HTTP* odgovorom koji sadrži traženu listu, koju klijent prikazuje u dialogu za otvaranje datoteka.

Kada korisnik zatraži snimanje trenutnog stanja u datoteku, upućuje se `HTTP POST` metoda poslužitelju *RepoUpload* s trenutnom korisničkom oznakom, imenom datoteke i reprezentacijom konteksta. Kada poslužitelj zaprimi takav zahtjev, koristeći primjerene apstrakcije generira jedinstvenu reprezentaciju primljenih podataka te od *Google App Engine* sustava zatražuje kontekst koji sustav već čuva za danog korisnika kojemu dodaje novozaprimljenu datoteku. Tako proširen kontekst se ponovno snima u trajno spremište trenutnog korisnika, a klijentu se vraća odgovor kako je snimanje uspješno. U slučaju neočekivane pogreške, klijent prima opis iznimke koja se dogodila.



Slika 4.3. Arhitektura primjenskog programa

5. Programsko ostvarenje korisničkog sučelja

U ovom poglavlju opisuje se korisničko sučelje Toolbox udomljenika, njegova organizacija i programsko ostvarenje.

Struktura udomljenika opisana je u poglavlju 5.1. Donosi se opis *XML* specifikacije udomljenika i njegov izgled. Prikazana je sinkronizacija prikaza i opis programskog ostvarenja model-prikaz-upravljač paradigme (poglavlje 1.1).

Poglavlje 5.2 iznosi sinkronizaciju prikaza Toolbox udomljenika i programsko ostvarenje paradigme model-prikaz-upravljač.

5.1. Prezentacijski sloj udomljenika

Toolbox udomljenik oblikovan je u *HTML* jeziku koji se dinamički poslužuje preko *Google Gadget* sustava (poglavlje 2.6). Zajedno s definicijom odnosa web elemenata, sustav poslužuje i opis izgleda elemenata u *CSS* standardu kojim se predodžbeno oblikuju elementi primjenskog programa.

Sustav udomljenika poslužuje primjenski program unutar zadanog okvira prema *HTML* standardima u okruženju kao što je *iGoogle* ili *orkut*, no moguće je posluživati udomljenik na bilo kojoj stranici koja ima pristup poslužitelju, osiguravajući dostupnost i točnost prikaza neovisno o mogućne neprimjerenom standardu korištenom na stranici.

5.2. Sinkronizacija prikaza

Sinkronizacija prikaza čest je problem u dinamičnim web primjenskim programa koje na isti *model* podataka žele prikazati na više načina (*pogleda*) tako da svaki pogled na model daje ažurnu prikazivanje određenog podskupa podataka koje su u modelu. Nadalje, svaka korisnikova promjena podataka se mora propagirati kroz sve pogleda tako da bi se osigurala ažurnost.

5.2.1. Sinkronizacija Toolbox udomljenika

U slučaju Toolbox udomljenika, model podataka predstavljaju sve informacije o elementima koje su na radnoj površini: njihovi tipovi (gumb, polje za unos teksta, lista, ...), stilovi (boja, veličina, granice, ...) i ostale informacije. Jedan od mogućih pogleda na model bila bi lista svih svojstava i njihovih vrijednosti, no takav prikaz nije praktičan za korisnika koji uređuje elemente. Pogled kroz koji radna površina prikazuje podatke iz modela jednostavno iscrtava element sa svim postavkama za svaki zapis u modelu, dodatno osluškujući događaje od strane korisnika (zahtjev za brisanjem elementa, zahtjev za uređivanje izgleda elementa, ...), dok pogled u traci za uređivanje postavki prikazuje sva svojstva trenutno odabranog elementa (pogled je filtriran odabranim elementom). Treći prikaz jednostavno enumerira sve elemente u padajućem izborniku pod nazivom tipa elementa i indeksom (npr. `TextArea (id = 0)`), ignorirajući sva ostala svojstva elementa.

Promjeni li korisnik neko od svojstava elementa ili ga pak potpuno izbriše sa radne površine, model mora reflektirati te promjene, a slijedno i svi ostali pogledi; izbriše li se element čija se svojstva prikazuju u traci za svojstva, traka se mora ukloniti. Slično, u padajućem izborniku se treba ukloniti izbrisani element, a na radnoj površini se ne smije više iscrtavati. Promjeni li se veličina, boja ili neko drugo svojstvo koje direktn utječe na izgled elementa, mora se promijeniti i iscrtani element kako bi pratio te promjene. Kako bi se postigla međusobna komunikacija modela i svih pogleda, definira se upravljač (engl. *controller*) koji vrši sinkronizaciju.

Sinkronizacija prikaza je ostvarena model-prikaz-upravljač paradigom opisanom u poglavlju 1.1. Upravljač obrađuje i reagira na sve moguće događaje, te o tome obavještava sve sudionike, pa ako korisnik odabere brisanje elementa, taj se događaj od pogleda prosljeđuje upravljaču koji prvo obavještava model da ukloni element, a potom nalaže svim pogledima da se ponovno iscrtaju koristeći svježe podatke. Isti se proces izvršava za bilo kakav tip promjene.

metoda	opis
`getDataByIndex(index)`	dobavlja element određen indeksom elementa u modelu ako postoji, inače oznaku za nepostojeći element
`addItem(item)`	dodaje element u model ako već nije u modelu
`removeItem(index)`	ako element postoji u modelu, briše element iz modela

Tablica 5.1. Metode nad modelom (kroz upravljač)

metoda	opis
`addListener(view)`	ovom metodom se pogled prebilježava kod upravljača da želi primati obavijesti o promjenama modela
`removeListener(view)`	ako registrirani pogled ne želi više primati obavijesti o promjenama modela, na ovaj način se može odjaviti
`triggerUpdate()`	upravljač poziva svaki pretplaćeni pogled da se osvježe kako bi se sinkronizirali sa podatcima u modelu

Tablica 5.2. Metode nad upravljačem

5.2.2. Ostvarenje paradigme model-prikaz-upravljač

Programsko ostvarenje model-prikaz-upravljač uzorka za udomljenik izvedena je u *JavaScript* skriptnom programskom jeziku, a očekivano se sastoji od modela, pogleda i upravljača.

Model se sastoji od objekta `data` koji čuva podatke o elementima i indeksa trenutno odbranog elementa na radnoj površini (`selected`) koji je na početku postavljen na -1 kako bi označio da niti jedan element nije odabran. Nad modelom je moguće obavljati tri akcije dane tablicom 5.1.

Kako se radi o *web* primjenskom programu, svaki pogled mora biti dio *HTML* stranice. Kad god se podaci promijene, upravljač mora osvježiti pogled, tj. zadati mu da se ponovno iscrta u pregledniku. Svakom pogledu dodjeljen je *web* element `targetControl` u kojem se iscrtava, te funkcija `drawMethod` koja određuje način na koji se iscrtava, tj. koje podatke iz modela prikazuje i na koji način. Zada li upravljač pogledu da se osvježi, nad pogledom se izvrši metoda `redraw` koja osvježi prikaz i sinkronizira stanje primjenskog programa. Upravljač se može logički podijeliti na dva dijela, i to na dio koji upravlja osvježavanjem prikaza te na dio koji reagira na korisnikove akcije i uređuje model.

Pregled komunikacije upravljača sa prikazima prikazan je u tablici 5.2. Početni koraci u ostvarenju paradigme su stvaranje objekata modela, prikaza i upravljača, kao što je prikazano u ispisu 5.1.

```
model = new gpHC.MVCLibrary.Model([])
controller = new gpHC.MVCLibrary.Controller(model);

viewPanel = new gpHC.MVCLibrary.View(controller.model,
_gel("workBoxContainer"), function(targetControl) {...});

viewList = new gpHC.MVCLibrary.View(controller.model,
_gel("p_current_control"), function(targetControl) {...});

viewProperties = new gpHC.MVCLibrary.View(controller.model,
_gel("properties_grid_span"), function(targetControl){...});
```

Ispis 5.1. Stvaranje objekata MVC paradigme

U procesu stvaranja objekata prvo se stvori model koji sadrži praznu listu podataka, te se stvori upravljač za taj model. Nakon toga, određuju se tri prikaza sa kojima se poveže upravljač, te im se preda *DOM* element u kojoj će se iscrtavati i ostvarenje funkcije iscrtavanja koju upravljač poziva kada se model promijeni. Dok radna površina koristi kompliciraniju funkciju za iscrtavanje prikaza koja prolazi kroz cijeli model i za svaki element u njemu stvara *HTML* element zadanog tipa i raščlanjujući model, postavlja atribute i postavke novostvorenog elementa tako da odgovaraju modelu te finalno koristeći *DOM* funkcionalnost umetne element, funckija za prikaz odabranog elementa u padajućem izborniku pobriše sve svoje elemente te za osvježeni model za svaki element dodaje odgovarajući unos vodeći računa pritom da ako indeks elementa koji se treuntno dodaje odgovara indeksu odabranog elementa u osvježenom indeksu, postavi vrijednost atributa `selected` na istinitu vrijednost.

Raščlanjivanjem datoteke za specifikaciju *HTML* elemenata, gradi se polje elemenata `DataElements` u kojeg se dodaje prototip svakog zadanog elementa sa početnim vrijednostima atributa, kao što je prikazano u ispisu 5.2.

```
DataElements = new Array();
for(var i=0, elements = xmlDoc.getElementsByTagName("element");i<
    elements.length;++i) {
  var current_element = new HTML_Element(parse(elements[i]));
  DataElements.push(current_element);
}
for(var i=0; i<DataElements.length;++i) {
  var innerA = document.createElement("a");
  innerA.innerHTML = DataElements[i].toToolboxItem();
  $("parent").appendChild(innerA);
}
```

Ispis 5.2. Izrada alatne trake

Kada se element iz alatne trake doda na radnu površinu, poziva se metoda za dodavanje elemenata nad upravljačem, koji tada doda element u model te obavijesti sve prikaze da se osvježe, kao što je prikazano na ispisu 5.3. Metoda *drag-and-drop* koristi se pomoću sučelja `_IG_Drag` koji je dio sustava Google udomljenika na način da se svaki element u radnoj traci naznači kao izvor, tj. element koji se može vući, a radna površina kao cilj, tj. element u koji se može ispustiti drugi element. Kada se element ispusti, sučelje podiže `onDragEnd` događaj, te se prototip elementa koji je ispušten klonira, a novodobijeni element doda u model metodom upravljača.

```
var drag_obj = new _IG_Drag();
drag_obj.addTarget("workBox_div", _gel("workBoxContainer"));

drag_obj.onDragEnd = function(source, target) {
  var i = parseInt(source.getAttribute("tag"));
  var addedElement = DataElements[i].clone();

  controller.addItem(addedElement);
}
```

Ispis 5.3. Dodavanje elemenata

Brisanje elemenata vrši se pritiskom na sliku za brisanje elementa u radnoj površini. Slika je postavljena tako da se klikom na nju obaviještava upravljač kako se zadani element briše, što nadalje rezultira time da nakon što osvježi model, upravljač svakom pretplaćenom prikazu šalje signal da se osvježe, kao što je prikazano u ispisu 5.4.

```
deleteImage.setAttribute("src", "delete.gif");
deleteImage.setAttribute("title", "Delete this control.");
deleteImage.setAttribute("onclick", "javascript:removeElement("+i+")");

...

removeElement = function (i) {
  controller.removeItem(i);
}
```

Ispis 5.4. Brisanje elemenata

Najopsežnije oblikovanje postavki događa se pri uređivanju `style` atributa, kada se iz mnoštva elemenata raščlanjuju vrijednosti u jedinstvenu vrijednost atributa stila, te se nad nad zadanim elementom poziva osvježenje svojstva. Svojstvo stila sadrži preko dvadeset parova ključ/vrijednost koje oreďuju segmente izgleda elementa.

5.3. Posluživanje HTML elemenata

Specifikacija *HTML* podržanih elemenata čuva se u *XML* obliku na poslužitelju. Primjenski program pri inicijalizaciji dohvaća specifikaciju te na osnovi sadržaja nudi korisniku korištenje podržanih elemenata. Za neke od atributa se zahtjeva da budu određenog tipa, te je za svaki atribut elementa moguće specificirati očekivani oblik vrijednosti atributa. Unošenjem vrijednosti atributa, dinamično se provjerava valjanost unesene vrijednosti.

5.3.1. Opis HTML elemenata

Svaki *HTML* element može sadržavati niz atributa koji su stanadardno definirani za svaki element u specifikaciji *HTML* jezika. Postoji skup atributa koje podržava više *HTML* elemenata kao što su `id`, `tag` ili `name`, no postoje i određeni atributi koji imaju smisleno značenje samo za određene elemente, kao što su rukovoditelji događaja `onclick` i `onfocus` koji opisuju događanja u slučaju da se dogodi određeni događaj.

Prema *HTML* standardu, atributi elemenata zapisuju se kao dio elementa u parovima ključ - vrijednost, kao što je prikazano u ispisu 5.5. Svaki par ključ - vrijednost određuje jedno svojstvo *HTML* elementa koje se kasnije može koristiti

od strane korisnika ili pak preglednika.

```
<a href="http://www.fer.hr/"
   id="poveznica"
   alt="Pritisni me"
   title="FER Online"
   onfocus="dobioFokus()"
   charset="utf-8">Poveznica</a>
```

Ispis 5.5. Primjer opisa *HTML* elementa atributima

Opis *HTML* elemenata koje nudi *Toolbox* udomljenik zapisan je u *XML* obliku kao što se vidi u ispisu 5.6. Element za unos teksta opisan tako da je uz njegov opis i ključnu riječ `textarea` stoji i niz svih mogućih atributi tj. imena, opisa i tipova podataka. Dodavanje novih elemenata u *Toolbox* udomljenik svodi se na zapis definicije tog elementa i njegovih atributa u prikazanom *XML* formatu u zadanu datoteku na poslužitelju.

```
<?xml version="1.0" encoding="UTF-8"?>
<html_elements>
  <element id="TextArea" desc="The textarea element is used to create a
      multi-line text input form control." tag="textarea">

    <property regex="/^\d*$/" desc="(Number) This attribute specifies
       the visible width in average character widths. This attribute
       is required.">cols</property>

    <property regex="/[A-Za-z_0-9]*/" desc="(Text) Form control name.">
       name</property>

    <property regex="/^\d*$/" desc="(Number) This attribute specifies
       the number of visible text lines. This attribute is required.">
       rows</property>
    ...
  </element>
  ...
</html_elements>
```

Ispis 5.6. Primjer opisa *HTML* elementa *Toolbox* udomljenika

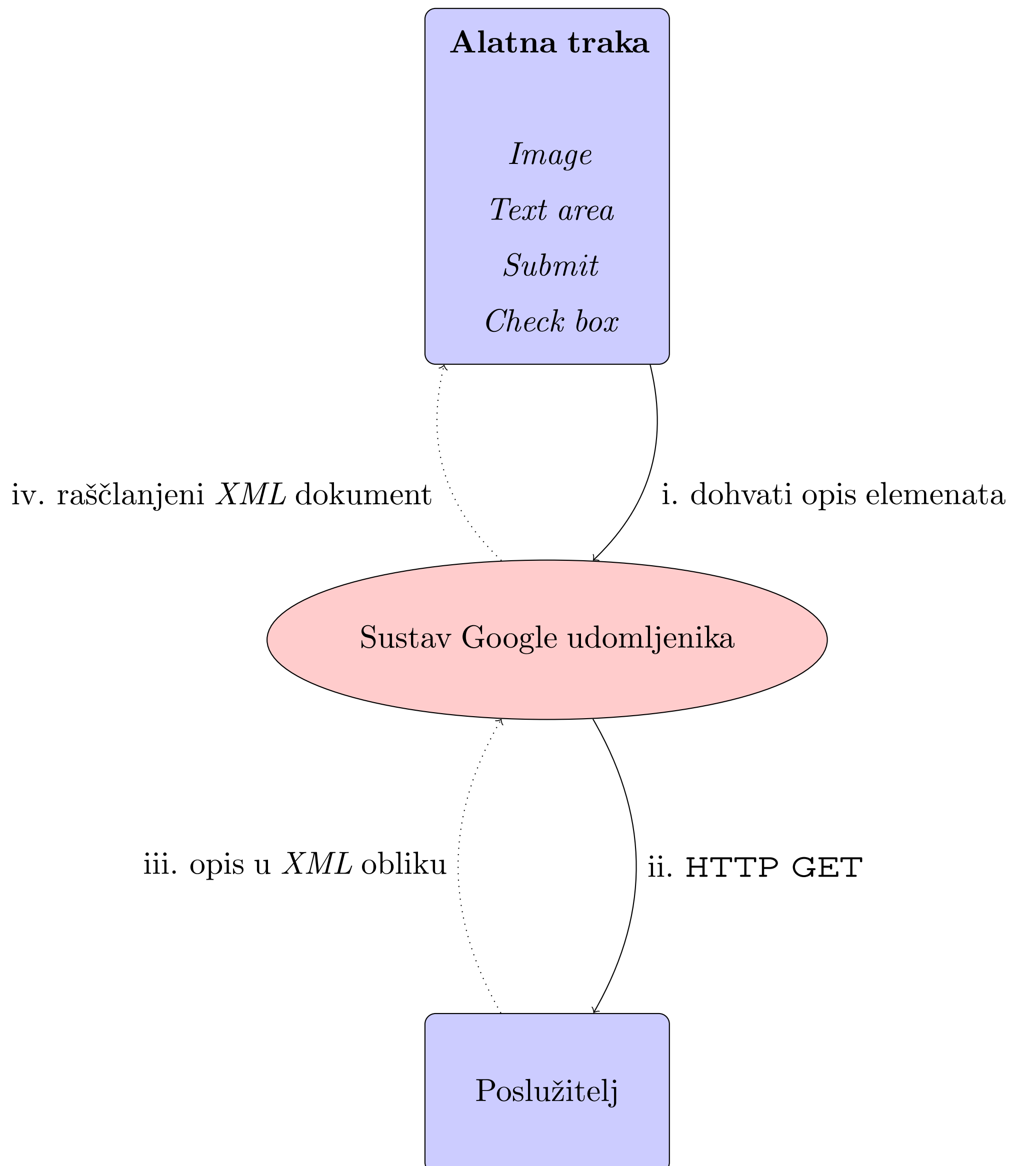
Atribut `regex` ima posebno značenje koje nije vezano uz *HTML* standard već služi za provjeru točnosti oblika podatka atributa. Element za upis teksta, primje-

rice, ima atribut `rows` koji definira koliko je vidljivih linija prikazano, i taj atribut mora imati cjelobrojnu vrijednost, dok su sve druge vrijednosti netočne i neprimjerene za taj atribut. *Toolbox* udomljenik provjerava pri unosu i promjeni svakog atributa jeli novounesena vrijednost primjerena atributu, tj. odgovara li regularnom izrazu zadanom `regex` atributom u opisu elementa i ne dopušta unos krive vrijednosti.

5.3.2. Dinamično dobavljanje opisa elemenata

Kako je opis *HTML* elemenata potreban primjenskom programu da bi učitao korisničko sučelje i stvorio listu mogućih izbora, potrebno ga je dohvatiti pri učitavanju. Poželjno je da dohvat opisa ne uzrokuje blokiranje korisničkog sučelja primjenskog programa kako bi se poboljšao dojam korisnika, te je potrebno poslati asinkroni zahtjev na poslužitelj. Definicija *HTML* elemenata dana je u *XML* obliku što implicira da će po primitku opisa biti potrebno gramatički raščlaniti primljenu strukturu i od nje načiniti model za korisničko sučelje.

U poglavlju 2.6.2 opisan je imenski prostor gadgets.io koji pruža mogućnost korištenja ugrađene makeRequest funckije za slanje *HTTP* zahtjeva i obradu zaprimljenog odgovora od poslužitelja. Dobavljanje opisa elemenata vrši se slanjem *HTTP GET* zahtjeva na poslužitelj sa zadanom adresom specifikacije na poslužitelju i obradom odgovora. Kako moderni pretraživaći moraju gramatički raščlanjivati *HTML* i *XML* dokumente kako bi prikazali stranice, u sebi već sadržavaju raščlanjivać (engl. *parser*) koji podržava obilaženje i pretraživanje kako *DOM* stabla tako i bilo kojeg slično ustrojenog dokumenta kao što je definicija elemenata Toolbox udomljenika. Zaprimljeni se odgovor od poslužitelja ugrađenim mehaniznom raščlanjuje na objekte koji predstavljaju*HTML* elemente s njihovim atributima. Takvi objekti su glavna jedinica spremanja podataka o elementima u Toolbox udomljeniku, te se spremaju u model elemenata. Slika 5.1 prikazuje proces dobavljanja opisa *HTML* elemenata koji se sastoji od četiri koraka. U prvom koraku sučelje izdaje zahtjev za dohvat opisa elemenata (i). U drugom koraku, sustav Google udomljenika šalje poslužitelju `HTTP GET` zahtjev za specifikaciju elemenata (ii). U trećem koraku, poslužitelj odgovara sustavu udomljenika opisom elemenata u *XML* obliku (iii). U četvrtom koraku udomljenik raščlanuje zaprimljeni opis i stvara alatnu traku na temelju dobivene specifikacije (iv).



Slika 5.1. Dobavljanje opisa *HTML* elemenata

Po završetku obrade zaprimljenog opisa elemenata, udomljenik iscrtava alatnu traku koja sadrži svaki opisani element s odgovarajućim opisom. Primjenski program pri pokretanju dobavlja svježi opis elemenata u slučaju da je u međuvremenu promjenjen, osiguravajući tako ažurnost podataka.

6. Programsko ostvarenje poslužitelja

Zadaća poslužitelja je pouzdano posluživanje spremljenih datoteka i spremanje novih opisa konteksta. Sloj za spremanje konteksta opisan je u poglavlju 6.1, dok je sustav autentikacije izložen u poglavlju 6.2. Opis komunikacije s repozitorijom iznesen je u poglavlju 6.3.

6.1. Struktura sloja za spremanje konteksta

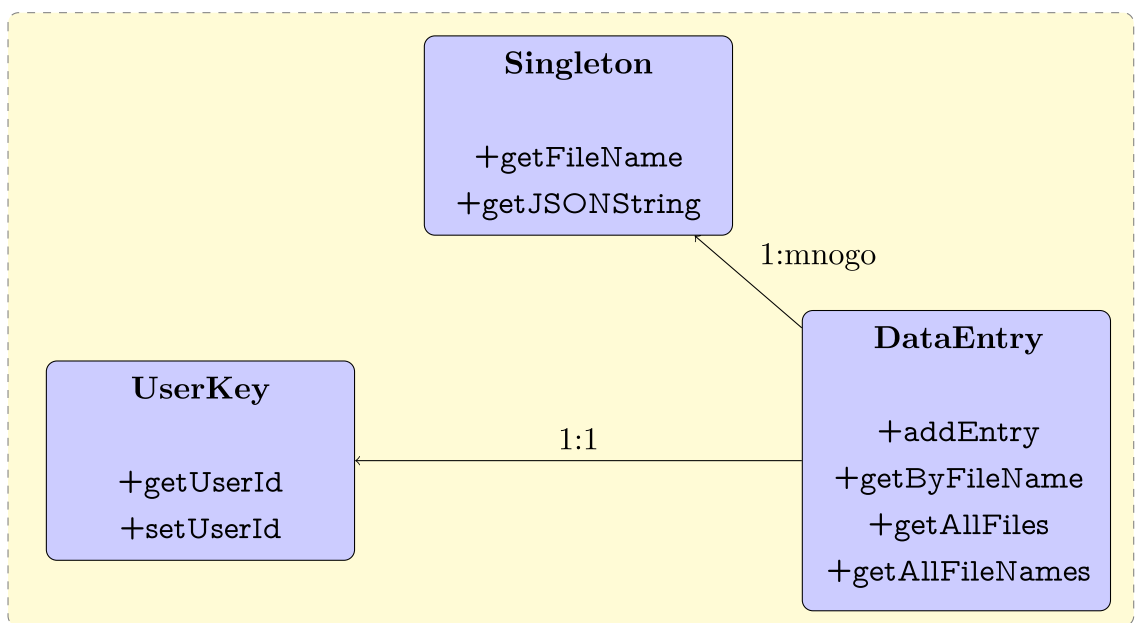
Sloj za spremanje konteksta sadrži definiciju funckionalnog repozitorija sposobnog za spremanje, dobavljanje i brisanje podataka, a sastoji se apstrakcije za korisnika udomljenika, apstrakcije za repozitorij jednog korisnika tj. niza datoteka, apstrakcije za jednu datoteku koja se sprema u repozitorij, te dvije službe koje odgovaraju na korisnikove zahtjeve za upravljanjem podataka.

Koristeći pribilješke (engl. *annotation*) Java podatkovih objekata, za razred se određuje način na koji će biti serijaliziran u trajnu pohranu, određujući tako primarni ključ spremanja i polja koja se trebaju spremiti, kao što je prikazano u ispisu 6.1.

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class UserKey {
    @PrimaryKey private String id;
    @Persistent private long userID=0
    ...
}
```

Ispis 6.1. Primjer pribilješki *JDO* sučelja

Arhitektura sloja dana je slikom 6.1.



Slika 6.1. Struktura za spremanje konteksta

6.1.1. Razred *UserKey*

Razred `UserKey` predstavlja korisnika u bazi podataka. Korisnik se identificira svojim ključem, koji mu poslužitelj pridjeljuje ako ga već nema, a sprema se u udomljeniku kao korisnička varijabla, kako je opisano u poglavlju 2.6.

Metode nad prikazom korisnika dane su tablicom 6.1.

metoda	opis
`dohvati ključ korisnika`	razred vraća ključ trenutnog(ili novog) korisnika

Tablica 6.1. Metode nad `UserKey` tipom

6.1.2. Razred *DataEntry*

Razred `DataEntry` predstavlja objekt koji sadrži cijeli repozitorij za jednog korisnika i osigurava apstrakciju baze podataka; ono što se nalazi u ovom objektu je u bazi podataka. Objekt nakon svake promjene osvježi bazu podataka zato jer je registriran kod *Persistance Manager* službe kao objekt baze podataka. Repozitorij jednog korisnika opisan je nizom imena datoteka i njihovih pripadajućih sadržaja. Metode nad ovim tipom opisane su tablicom 6.2.

metoda	opis
`dohvat datoteke`	razred vraća sadržaj tražene datoteke
`brisanje datoteke`	razred briše zadanu datoteka iz svog konteksta
`dohvat popisa datoteka`	razred vraća popis datoteka trenutno u repozitoriju
`dohvat datoteke po imenu`	razred vraća sadržaj zadane datoteke

Tablica 6.2. Metode nad `DataEntry` tipom

6.1.3. Razred *Singleton*

Razred `Singleton` apstrahira jednu datoteku u bazi podataka. Razred `DataEntry` sprema datoteke upravo kao listu `Singleton` objekata vezane za trenutnog korisnika. Metode dostupne nad ovim tipom dane su tablicom 6.3.

metoda	opis
`dohvat ime datoteke`	razred vraća ime datoteke koju predstavlja
`dohvat sadržaja datoteke`	razred briše sadržaj datoteke koju predstavlja

Tablica 6.3. Metode nad `Singleton` tipom

6.1.4. Razred *RepoFetch*

HTTP servlet `RepoFetch` osluškuje korisnikove zahtjeve za dohvat podatka. Korisnik može zahtijevati različite informacije od poslužitelja, kao što je opisano u tablici 6.4. Jednom serijaliziranom podatkovnon objektu može se pristupiti korištenjem *Persistance* sučelja. Poznavajući ime razreda traženog objekta i ključ pod kojim je spremljen, moguće je pristupiti spremljenom objektu kao što je prikazano na ispisu 6.2. Dohvaćeni objekt može se direktno mijenjati; sučelje ga poznaje kao objekt vezan za trajnu memoriju te prati promjene koje se vrše nad njim.

```java
PersistenceManager pm = PMF.get().getPersistenceManager();
DataEntry podaci = pm.getObjectById(DataEntry.class, user);
```

Ispis 6.2. Primjer dohvata spremljenog *JDO* objekta

metoda	opis
dohvat liste datoteka	*servlet* traži od baze podataka listu svih datoteka spremljenih za trenutnog korisnika i vraća ih klijentu
dohvat datoteke	*servlet* traži od baze podataka sadržaj tražene datoteke spremljene za trenutnog korisnika i vraća je klijentu
brisanje datoteke	*servlet* briše iz baze podataka traženu datoteku spremljenu za trenutnog korisnika

Tablica 6.4. Metode nad `RepoFetch` tipom

6.1.5. Razred *RepoUpload*

HTTP servlet `RepoUpload` osluškuje korisnikove zahtjeve za spremanje podataka. U slučaju da korisnik želi pohraniti kontekst u repozitorij, mora koristiti metodu danu tablicom 6.5. Spremanje podataka na repozitorij iziskuje spremanje objekta u trajnu memoriju korištenjem *Persitance* sučelja. Dovoljno je registrirati objekt kod *Persitance manager* sučelja kao objekt koji se želi spremiti, kao što je prikazano na ispisu 6.3.

```java
PersistenceManager pm = PMF.get().getPersistenceManager();
DataEntry podaci = new DataEntry(new UserKey("korisnik"));

pm.makePersistent(podaci); // registracija
podaci.addEntry("serijalizirana datoteka", "ime datoteke");
pm.close();
```

Ispis 6.3. Primjer spremanja objekta u trajnu memoriju

metoda	opis
`pohrani kontekst`	*servlet* sprema u bazu podataka kontekst dostavljen od klijenta za trenutnog korisnika

Tablica 6.5. Metode nad `RepoUpload` tipom

6.2. Autentikacija korisnika

U današnjim primjenskim programima sigurnost podataka vrlo je bitna. U slučaju repozitorija, svaki korisnik ima svoj podskup datoteka koje su pohranjene na poslužitelju i koje može dohvatiti, no sa strane poslužitelja, radi se o jednom velikom distribuiranom skladištu. Kako bi se razlikovali korisnici, potrebno im je predati dodijeliti ključ te ih njime identificirati.

6.2.1. Dodjeljivanje identifikacijskog broja

Kada do sada repozitoriju nepoznat korinsik zatraži pohranjivanje datoteke, poslužitelj klijentu šalje i jedinstveni ključ (redni broj korisnika) za korisnika i od tada nadalje taj ključ veže za trenutnog korisnika. Klijent sprema ključ u memoriju udomljenika, i u svaki sljedeći zahtjev umeće identifikacijski broj kako bi se predstavio poslužitelju.

Ako korisnik već ima svoj identifikacijski broj, to znaći da mu je već bio dodjeljen od strane poslužitelja i spremljen unutar memorije udomljenika.

6.2.2. Provjera identifikacijskog broja

Kada repozitorij zaprimi zahtjev, a identifikacijski broj korisnika je postavljen na posebnu vrijednost koja identificira nepoznatog korisnika, uz odgovor klijentu šalje novi identifikacijski broj. Ako je zaprimjen zahtjev od već registriranog korisnika, njegov identifikacijski broj se koristi kao ključ za dobavljanje i spremanje podataka u bazu podataka, osiguravajući time odvojenost i privatnost podataka različitih korisnika.

6.3. Zahtjevi prema poslužitelju

Spremanje konteksta u repozitorij vrši se serijalizacijom Java podatkovih objekata opisanih u poglavlju 2.7 i spremanjem u trajnu bazu podataka *App Engine* sustava koji se naziva *datastore*. Mogući zahtjevi su spremanje, dobavljanje i brisanje konteksta.

6.3.1. Obrada zahtjeva za spremanje konteksta

Zahtjev za spremanje konteksta je *HTTP POST* zahtjev koji sadrži tri dodatna parametra, i to ime datoteke koja se sprema, njen tekstualni sadržaj te identifikacijski broj aktivnog korisnika.

Kada poslužitelj zaprimi zahtjev za spremanje konteksta, kontekst se kao popis datoteka i njihovih sadržaja vezanih za trenutnog korisnika sprema u trajnu memoriju pod ključem korisnikovog identifikacijskog broja. Ako je za trenutnog korisnika već postojao spremljen kontekst, on se učivata iz trajne memorije i nadopunjava novododanim zapisom i ponovno sprema u bazu podataka.

U slučaju da se radi o novom korisniku, tj. za njega ne postoji unos u bazi podataka, sustav generira novi identifikacijski broj koji prosljeđuje klijentu uz odgovor.

6.3.2. Obrada zahtjeva za dobavljanje konteksta

Zahtjev za spremanje konteksta je *HTTP GET* zahtjev koji sadrži može biti dvojak: zahtjev za popis datoteka određenog korisnika ili zahtjev za sadržaj određene datoteke korisnika. Uz zahtjev, šalje se i identifikacijski broj korisnika. Ako je specificirano ime datoteke, poslužitelj dohvaća iz trajne memorije trenutnog korisnika `DataEntry` objekt i u njemu pronalazi traženu datoteku, čita je, te zapisuje sadržaj u odgovor koji će vratiti klijentu. U slučaju da ime tražene datoteke nije određeno, poslužitelj dohvaća popis svih mogućih datoteka i šalje ga klijentu.

U slučaju da zapis za korisnika ili datoteku ne postoji, poslužitelj obaviještava klijent kako je došlo do greške u obradi zahtjeva.

6.3.3. Obrada zahtjeva za brisanje konteksta

Zahtjev za brisanje konteksta je *HTTP GET* zahtjev koji sadrži dva dodatna parametra, i to identifikacijski broj trenutnog korisnika te ime datoteke koju je

potrebno izbrisati.

Poslužitelj dohvaća iz trajne memorije za trenutnog korisnika zapis repozitorija, te nad njim poziva metodu za brisanje datoteke, dajući ime kao parametar. Kako se objekt nalazi smotrom *Persistance* sučelja koje prati sve izvršene promjene, nikakve daljne akcije nisu potrebne kako bi se stanje propagiralo u trajnu memoriju. Primjer brisanja zapisa datoteke dan je u ispisu 6.4.

```
PersistenceManager pm = PMF.get().getPersistenceManager();
DataEntry podaci = pm.getObjectById(DataEntry.class, "korisnik");

podaci.removeEntry("ime datoteke");
```

Ispis 6.4. Primjer brisanja datoteke iz trajne memorije

Zaključak

Razvojem Interneta kao platforme došlo je do poboljšanja korisničkog iskustva na mreži, no i do složenijeg procesa ostvarivanja mrežnih primjenskih programa. Sustavi su kompleksniji i masivniji, moraju podržavati milijune korisnika i garantirati dostupnost, pravovremenost i sigurnost njihovih podataka. Korištenje paradigme model-prikaz-upravljač u web orkuženju česta je pojava, no u raspodijeljenim potrošaćkim primjenskim programima sve više i standard. Osim logičke odvojivosti koje paradigma nalaže, u današnje vrijeme iznimno je važno ispitivanje sustava.

Uz primjere u ovom radu, prikazane su osnovne tehnologije i principi razvoja raspodijeljenih sustava na *Google* infrastrukturi. Za potrebe rada, razvijen je udomljenik za uređivanje *HTML* elemenata i njihovo ponovno korištenje u oblikovanju drugih udomljenika. Uz udomljenik, razvijen je raspodijeljeni repozitorij koji udomljenik koristi kao spremište podataka, a može istovremeno posluživati veći broj korisnika. Udomljenik se može ugraditi u bilo koju web stranicu ili pak radnu površinu korisnikovog operacijskog sustava. Ukoliko korisnik nema pristup mreži, udomljenik se može udomiti u *Google Desktop* sustav i tako koristiti, no u tom slučaju repozitorij nije dostupan, što ne narušava, već ograničava rad primjenskog programa.

Važnost *Toolbox* udomljenika ne leži toliko u njegovoj funkcionalnosti koliko u prilagodljivosti okolini i jednostavnosti korištenja. Razvoj web korisničkog sučelja sveden je na nekoliko klikova miša, a potrebna okolina za rad smanjena je na Internet vezu i web preglednik – zahtjeve koje ispunjava svaki moderniji mobilni telefon.

Goran Petrović

Literatura

[1] Google App Engine: App Engine Java Overview. Google Code, 2009. [online] http://code.google.com/appengine/. 27

[2] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0. *W3C recommendation*, 6, 2000. 18

[3] M. Broy and Aleksandr Vasilevich Zamulin. *Perspectives of system informatics*. Springer, 2003. 4

[4] E. Ciurana. *Developing with Google App Engine*. Apress, 2009. 27

[5] D. Crockford. *JavaScript: the good parts*. O'Reilly Media, Inc., 2008. 13, 17

[6] R. Fielding, J. Gettys, JC Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol-HTTP/1.1. 1999. 20, 21

[7] D. Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., 2006. 14

[8] David Flanagan. *JavaScript: The Definitive Guide, Fourth Edition*. O'Reilly Media, fourth edition, 2001. 13

[9] Google Gadgets API: gadgets.* API Developer's Guide. Google Code, 2008. [online] http://code.google.com/apis/gadgets/. 23

[10] J.J. Garrett et al. Ajax: A new approach to web applications. 2005. 22

[11] Jason Hunter and William Crawford. *Java Servlet Programming*. O'Reilly Media, second edition, 2001. 28

[12] A. Leff and J.T. Rayfield. Web-application development using the model/view/controller design pattern. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, page 118. IEEE Computer Society, 2001. 4

[13] M. Mahemoff. *Ajax design patterns.* O'Reilly Media, Inc., 2006. 22

[14] J. Marini. *Document Object Model.* McGraw-Hill, Inc. New York, NY, USA, 2002. 9

[15] E.A. Meyer. *Cascading style sheets: the definitive guide.* O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2004. 33

[16] J. Mitchell-Wong, R. Kowalczyk, A. Roshelova, B. Joy, and H. Tsai. Opensocial: From social networks to social ecosystem. In *Inaugural IEEE-IES Digital EcoSystems and Technologies Conference, 2007. DEST'07*, pages 361–366, 2007. 25

[17] C. Musciano and B. Kennedy. *HTML & XHTML: the definitive guide.* O'Reilly Media, Inc., 2006. 9

[18] L.D. Paulson. Building rich web applications with Ajax. *COMPUTER-IEEE COMPUTER SOCIETY-*, 38(10):14, 2005. 39