VELEUČILIŠTE U RIJECI

Tin Crnković

Sljedeća generacija *web* aplikacija

(specijalistički završni rad)

Rijeka, 2013.

VELEUČILIŠTE U RIJECI

Poslovni odjel

Specijalistički diplomski stručni studij Informacijske tehnologije u poslovim sustavima

Sljedeća generacija *web* aplikacija

(specijalistički završni rad)

MENTOR
Mr. Sc. Marin Kaluža, *v. pred.*

STUDENT
Tin Crnković, *bacc. inf.*
MBS: 2422034734/10

Rijeka, svibanj 2013.

VELEUČILIŠTE U RIJECI
Poslovni odjel
Rijeka, 8.03.2013.

ZADATAK
za specijalistički završni rad

Pristupniku: Tin Crnković **MBS: 2422034734/10**

Studentu specijalističkog diplomskog stručnog studija programskog inženjerstva u poslovnim sustavima, izdaje se zadatak za završni rad – tema specijalističkog završnog rada pod nazivom:

Sljedeća generacija web aplikacija

Sadržaj zadatka:

Objasniti višeslojne arhitekture web aplikacija. Pojasniti primjenu i mogućnosti tehnologija dostupnih po pojedinim slojevima aplikacije. Pokazati nove mogućnosti aplikacije i korisničkog sučelja koje donose nove tehnologije razvoja. Prikazane tehnologije primjeniti u izgradnji aplikacije. Izgraditi projektnu dokumentaciju i aplikaciju društvene mreže koristeći web framework, real-time web tehnologije, distribuiranu bazu podataka, JavaScript, HTML5 i CSS3. Usporediti prednosti i nedostatke prikazanih dostupnih tehnologija sa tradicionalnim pristupom razvoju web aplikacija.

Preporuka: _____

Rad obraditi sukladno odredbama Pravilnika o završnom radu Veleučilišta u Rijeci.

Zadano: 08.03.2013. **Predati do: 08.09.2013.**

Mentor:

_____

mr.sc. Marin Kaluža, v.pred.

Pročelnik odjela:

_____

mr.sc. Maja Gligora Marković, v.pred.

Zadatak primio dana: 08.03.2013

_____

Tin Crnković

Dostavlja se:
- **mentoru**
- **pristupniku**

I Z J A V A

Izjavljujem da sam specijalistički završni rad pod naslovom Sljedeća generacija web aplikacija izradio samostalno pod nadzorom i uz stručnu pomoć mentora mr. sc. Marina Kaluže.

Tin Crnković, *bacc. inf.*

_____

Sažetak

Specijalistički završni rad ‘Sljedeća generacija web aplikacija’ započinje kronološkim pregledom i usporedbom različitih arhitektura programskog rješenja, sa naglaskom na rasprostranjene arhitekture web aplikacija i trendove koji upućuju na daljnji smjer razvoja. Zatim se osvrće na upravljanje podacima i *hosting*, pri tome uspoređujući različite klijentske i poslužiteljske tehnologije pohrane prema kriterijima implementacije, performansi i mogućnosti migracije. Područje poslovnog sloja i komunikacije prikazano je kroz današnje stanje *web frameworka*, trendove *real-time web* komunikacije te JavaScript u smislu jezika, alata i zajednice na klijentu i poslužitelju. Sljedeće poglavlje prikazuje prezentacijski sloj modernih web aplikacija, te uključuje HTML5, CSS3, te CSS predprocesiranje. Rad se nastavlja izradom društvene mreže koristeći novonastale tehnologije i alate, te u konačnici ističe prednosti i nedostatke modernog razvoja web aplikacija u odnosu na tradicionalni pristup.

Ključne riječi: web aplikacija, ASP.NET MVC, JavaScript, node.js, HTML5, CSS3, LESS, SignalR, WebAPI, Azure, REST

SADRŽAJ

1. Uvod

"I travel not to go anywhere, but to go. I travel for travel's sake.
The great affair is to move."
- Robert Louis Stevenson

Svega desetak godina nakon pojave riječi *kiberprostor*[1], rađa se prva generacija digitalnih domorodaca: građana čije prirodno okruženje čine stalna mrežna povezanost i općedostupnost elektroničkih sadržaja i uređaja. Digitalna fotografija i videografija, društvene mreže i mrežne igre, tekstualna i video-komunikacija u stvarnom vremenu, pseudoanonimnost i alternativni identiteti, trenutne javne diskusije, *open courseware* i kriptovalute fenomeni su uz koje nova generacija umreženih korisnika odrasta, koristeći ih svakodnevno u privatne i poslovne svrhe.

Usprkos prvenstveno komercijalnim interesima vodećih pružatelja mrežnih usluga, Internet više nego bilo koji drugi novi javni resurs mijenja način na koji društvo komunicira, konzumira sadržaj i obavlja svakodnevne aktivnosti. Internet uvelike utječe na kvalitetu i stil života i postaje agent promjene u društvu. Iz navedenih razloga, područje web aplikacija i usluga postaje težište – točka zakretanja programskog inženjerstva.

Nova generacija korisnika postavlja pred struku brojne složene izazove, te uz razvoj novih tehnologija, alata, metodologija i uređaja udruženo stvara drugačiju sliku razvoja i funkcionalnosti web aplikacija u odnosu na rasprostranjene aplikacije osmišljene unazad nekoliko godina. Cilj ovog rada jest prikaz novih dostupnih tehnologija i alata prema pojedinim slojevima aplikacija, izrada aplikacije koristeći navedene resurse, kritički osvrt na postignute rezultate, te prikaz tehnoloških i arhitekturalnih trendova koji će nastaviti oblikovati web aplikacije u sljedećim generacijama.

---

[1] William Gibson osmislio je riječ *cyberspace* za kratku priču *Burning Chrome* 1982. godine, te ju popularizirao u romanu *Neuromancer* dvije godine kasnije

2. Arhitektura web aplikacija

"Perfection is achieved, not when there is nothing more to add,
but when there is nothing left to take away."
- Antoine de Saint-Exupéry

Softverska arhitektura je apstrakcija koja prikazuje strukturu programskog rješenja: način promatranja rješenja odnosno njegovih podsustava i njihovih međusobnih odnosa. Pojam arhitekture odnosi se na apstraktan pogled koji obuhvaća samo najvažnije čimbenike programskog rješenja, dok detalji implementacije i specifičnosti konačnog rješenja najčešće ostaju isključeni iz razmatranja do kasnijih faza razvoja. Prema Brooksu[2], arhitektura sustava predstavlja viziju njegove svrhe i način ostvarivanja iste.

U nastavku prikazujemo troslojnu arhitekturu kao predstavnika tradicionalnog načina razmatranja poslovnog informacijskog sustava, te zatim prikazujemo Model-View-Controller i Model-View-ViewModel arhitekture kao predstavnike suvremenijih arhitektura web i desktop aplikacija. Posljednja predstavljena arhitektura – kombinacija ranije navedenih arhitektura koju nazivamo mnogoslojnom, predstavlja sve češći slučaj u složenijim web aplikacijama u kojem tekuće potrebe i zahtjevi značajno mijenjaju arhitekturu tokom životnog ciklusa aplikacije.

2.1. Troslojna arhitektura

Troslojna arhitektura (en. *three-tier architecture*, *three-layer model*) je svaka arhitektura koja razlaže strukturu aplikacije na podatkovni, poslovni i prezentacijski sloj.

Podatkovni sloj (en. *data layer*, *data services layer*, *data access layer*) pruža usluge pristupa i manipulacije podacima, te upravlja njihovom stalnom ili privremenom pohranom.

Poslovni sloj (en. *business layer*, *middle tier*) provodi poslovna pravila, odnosno može ga se smatrati reprezentacijom svih pravila i okvira (poslovnih, legalnih, moralnih i drugih) unutar kojih se rješavaju problemi u domeni aplikacije. Poslovni sloj mora biti neovisan o ostalim

[2] Brooks, F., The Mythical Man-Month: Essays on Software Engineering, 1995.

slojevima; odnosno nastaviti provoditi poslovna pravila prilikom zamjene bilo kojeg drugog sloja. U primjere takve zamjene slojeva možemo ubrojiti migraciju klijenta, odnosno prezentacijskog sloja sa desktopa na web ili migraciju podataka, odnosno podatkovnog sloja sa vlastitog hardvera na dijeljenu cloud platformu. Usko vezan uz poslovni sloj je i pojam modela domene (en. *domain model*), odnosno konceptualnog modela kojim opisujemo poslovni problem čije je rješavanje cilj našeg sustava. Model domene opisuje entitete, atribute, uloge, veze i ograničenja poslovnog problema, odnosno okoline.

Prezentacijski sloj (en. *presentation layer*, *user services layer*) predstavlja korisničko sučelje putem kojeg se korisniku prikazuju podaci i omogućava manipulacija, odnosno putem kojeg se omogućuje interakcija korisnika i aplikacije.

Navedeni slojevi ne odgovaraju nužno različitim fizičkim lokacijama ili računalima, već predstavljaju samo logičku strukturu aplikacije. Troslojna arhitektura podrazumijeva određene preduvjete za uspješnu implementaciju:

- jasno definirana sučelja pojedinih slojeva aplikacije
- prezentacijski i podatkovni sloj međusobno smiju komunicirati samo posredstvom poslovnog sloja
- svaki sloj zahtjeva ekstenzivno nezavisno testiranje i vlastito rukovanje iznimkama (en. exception handling)

Iako koncipirana i prvi put implementirana sredinom devedesetih godina prošlog stoljeća (Donovan, MIT), troslojna arhitektura još je uvijek relevantna u kontekstu razvoja aplikacija jer pruža brojne prednosti kao što su strukturalna nezavisnost komponenti, fleksibilnost, lakoća održavanja, skalabilnost, ponovna iskoristivost i jasna podjela posla u funkcionalnom i implementacijskom smislu. Nedostatci troslojne arhitekture većinom nisu specifični, već opći nedostatci uvođenja dodatnih slojeva u programsku logiku: zahtjevniji proces identifikacije problema, povećanje početne kompleksnosti, veći broj unit testova zbog većeg broja komponenti i drugi.

2.2. Model-View-Controller

MVC (en. *Model-View-Controller*) je druga iznimno rasprostranjena softverska arhitektura koja dijeli aplikaciju na tri sloja: model, pogled (en. *view*) i kontroler.

Sloj modela zadužen je za podatke, ali i poslovnu logiku aplikacije: model enkapsulira poslovne entitete i izlaže sučelja koja omogućavaju korištenje. Uz poslovne podatke i entitete u strožem smislu, neke od uobičajenih klasa unutar modela zadužene su za pristup bazi podataka, provjeru identiteta korisnika, provjeru ispravnosti korisničkih unosa, slanje elektroničke pošte, enkripciju i slično.

Sloj pogleda zadužen je za prezentaciju sadržaja i uobičajeno sadrži predloške (en. *templates*) koje aplikacija koristi za prikaz podataka korisniku. Kontroler sloj je sučelje između modela i pogleda, on prikuplja korisničke podatke, instancira klase modela i poziva njegove javne metode. U tipičnom scenariju, kontroler će primiti korisnički zahtjev za sadržajem, instancirati model i pozvati njegove metode za dohvat sadržaja, smjestiti sadržaj u pogled i prikazati pogled korisniku.

Iako na prvi pogled slična troslojnoj arhitekturi, MVC arhitektura dodjeljuje različite odgovornosti svojim slojevima: pogledu nije omogućen izravan pristup funkcionalnosti poslovne logike, već se koristi kontroler kao posrednik. Također, sloj modela uz pristup podacima sadrži i poslovnu logiku, što ga čini svojevrsnom kombinacijom poslovnog i podatkovnog sloja troslojne arhitekture.

Većina dostupnih web frameworka danas koristi MVC kao podrazumijevanu arhitekturu, uključujući ASP.NET MVC, Django, Ruby on Rails, Zend, AngularJS, Locomitive, CakePHP, Struts i mnoge druge.

2.3. Model-View-ViewModel

MVVM (en. *Model-View-ViewModel*) je arhitekturalni obrazac koji uvodi novi posredni sloj između slojeva modela i pogleda: takozvani model pogleda (en. *view model*). Model pogleda predstavlja podatkovnu apstrakciju pogleda i sadrži sve podatke koje je potrebno prikazati

korisniku, ali ne i njihov izgled, što omogućuje korištenje drugog pogleda prilagođenog istoj klasi modela pogleda, ili korištenje drugog sloja modela koji generira istovrsni model pogleda. Sloj modela sličan je poslovnome sloju troslojne arhitekture, odnosno ne predstavlja samo podatke već i poslovnu logiku. MVVM olakšava *data binding*, odnosno sinkronizaciju podataka između pogleda i poslovnog sloja koja može biti jednostrana (na primjer, obavijest korisniku poslana u trenutku nastanka nekog događaja) ili obostrana (na primjer, višekorisnička kolaboracija u istovremenom stvaranju dokumenta, pri čemu se promjene sinkroniziraju od poslovnog sloja prema pogledu, ali i od pogleda prema poslovnom sloju).

Uvođenje dodatnog sloja separacije sadržaja i prezentacije olakšava izradu multi-platformskih aplikacija, poput web aplikacija prilagođenih različitim uređajima (pametni telefoni, tableti, desktop računala, *wearable computing* uređaji), a koristi se s tehnologijama kao što su HTML5, KnockoutJS, Windows Presentation Foundation i Silverlight. MVVM je relativno nova arhitektura, pa je tako jasna podjela posla i funkcionalnosti poput programske logike prikaza još uvijek predmet debate. Također, posebnu varijaciju MVVM arhitekture predstavlja uvođenje sloja kontrolera koji preuzima svu programsku logiku koja nije dio modela domene. Dodatnu prednost MVVM arhitekture uz visoku fleksibilnost platforme predstavlja i dodatno olakšanje programske logike iz sloja pogleda, što olakšava posao dizajnerima u stvaranju pogleda, ali i redizajnu aplikacije u kasnijem vremenskom periodu. Postojanje modela pogleda može olakšati *unit testing* proces jer je moguće automatizirati evaluaciju modela pogleda, odnosno podataka koje se putem pogleda isporučuje ili manipulira. Najčešća kritika MVVM arhitekture odnosi se na njenu neprilagođenost jednostavnijim UI rješenjima – odnosno uvjetima u kojima uvođenje velikog broja slojeva donosi relativno nesrazmjerno otežavanje izrade i održavanja bez značajnih prednosti. Iz tog razloga, MVVM se preporučuje samo u slučajevima velike raznolikosti ciljanih platformi ili velike vjerojatnosti zamjene slojeva, te pritom postojećoj relativnoj kompleknosti korisničkog sučelja.

2.4. Mnogoslojna arhitektura

Vodeći uzrok rastuće kompleksnosti modernih web aplikacija nalazimo u povećanju korisničkih zahtjeva i rastućim kriterijima ciljane grupe korisnika. Također, važan faktor čini veća zavisnost

javnih web aplikacija o drugim web aplikacijama i servisima, odnosno vanjskim uslugama i integracijama. U primjere takve vrste povećanja kompleksnosti ubrajamo integraciju društvenih mreža (Facebook for Websites, Twitter API, LinkedIn API, Google+ API, foursquare), vanjske usluge za autoriziju (OAuth, Google Identity Toolkit, Authy) i razne vanjske usluge specijalizirane za određeno područje zanimanja (Google AdWords, Google Maps, Google Calendar, YouTube, Instagram, Flickr, Tumblr, last.fm, SoundCloud, Grooveshark, Wikipedia, Yahoo Weather i brojni drugi).

Za pohranu podataka tradicionalno je zadužen sustav za upravljanje relacijskom bazom podataka, međutim novi trendovi ukazuju na raslojavanje pohrane na klijentsku i poslužiteljsku, koristeći razne relacijske i ne-relacijske strukture podataka i načine pristupa istima. Na klijentskoj strani, najrasprostranjeniji je tip pohrane riječnik, odnosno pohrana ključ-vrijednost parova (tehnologija Web Storage), dok neki preglednici omogućuju i pohranu indeksiranih redaka (IndexedDB) ili kompaktnu relacijsku bazu podataka (Web SQL). Na poslužiteljskoj strani, uz najrasprostranjeniju pohranu u relacijskim bazama, podaci se mogu pohranjivati u NoSQL tabličnim bazama, ili datotekama za pohranu generiranog sadržaja s ciljem poboljšanja performansi aplikacije (upotreba *caching* mehanizama).

Nedostatci JavaScript jezika, odnosno ECMAScript specifikacije i implementacije uzrokovale su pojavu brojnih naprednijih skriptnih jezika koje je moguće prevesti (kompajlirati) u JavaScript: CoffeeScript, TypeScript, JSX, Dart, Fantom i drugi prevode se na poslužiteljskoj ili klijentskoj strani u JavaScript kako bi ih web preglednik mogao izvoditi. JavaScript biblioteke također često preporučuju korištenje određene specifične arhitekture (poput MVC-a ili MVVM-a) u interakciji sa bibliotekom, pa tako možemo reći da postoji i 'arhitektura u arhitekturi' koja obuhvaća samo jedan segment cjelokupne web aplikacije. Nedostatcima Cascading Style Sheets stilskog jezika pristupilo se je na sličan način, pa se tako danas koriste Sass, LESS, Stylus i drugi napredni stilski jezici koji se prevode u CSS na poslužiteljskoj ili klijentskoj strani.

HTML5 i vezane specifikacije (Geolocation API, Web Storage, Web Sockets, WebGL, multimedijski sadržaji i druge.) implementirane su u različitoj mjeri i opsegu ovisno o tipu i inačici web preglednika. Iz toga razloga, web aplikacije moraju koristiti otkrivanje značajki (en.

feature detection) i omogućiti pristup alternativnom sadržaju ili funkcionalnosti u slučajevima u kojima web preglednik ne podržava traženu tehnologiju.

Razlike u uređajima korištenim za pristup web aplikacijama (mobilni telefoni, tablet računala, prijenosnici, stolna računala, pametni uređaji) upućuju na potrebu prilagodbe sadržaja vrlo različitim veličinama ekrana i uređajima za unos podataka, odnosno interakciju sa web aplikacijom. Iz tih razloga, responzivan dizajn (en. *responsive web design*) postaje nezaobilazan korak izrade modernih web aplikacija, dodatno povećavajući kompleksnost i produžavajući vrijeme razvoja.

Iz navedenih razloga postaje očito kako niti jedna ranije navedena arhitektura ne može prikazati i standardizirati funkcionalnost moderne web aplikcije, pa tako moderne web aplikacije možemo nazvati mnogoslojnim aplikacijama koje posjeduju amorfnu arhitekturu čija se struktura značajno mijenja tokom životnog ciklusa web aplikacije, a koja obuhvaća koncepte nekoliko tradicionalnijih ranije navedenih web arhitektura. Iako promjenjive arhitekture, moguće je navesti neka svojstva koja arhitektura, neovisno o konačnom obliku mora zadovoljavati:

- uvođenje novog sloja mora biti opravdano tekućim zahtjevima, ili jasnoj separaciji namjene
- izbjegavati uvođenje novih programskih ili skriptnih jezika, formata pohrane ili prijenosa podataka
- izbjegavati uvođenje novih protokola i poslužiteljske arhitekture
- jasno definirati subjekte kontrole i javna sučelja
- minimizirati rizik zaključavanja na određenu poslužiteljsku ili klijentsku platformu kao posljedicu arhitekturalne odluke
- izbjegavati potrebu ponavljanja ili međuovisnosti programske logike kao posljedicu arhitekturalne odluke

3. Podatkovni sloj i *hosting*

Nakon dugogodišnje dominacije relacijskih baza podataka, upravljanje podacima web aplikacije posljednjih je godina drastično izmjenjeno pojavom novih tehnologija kao što su NoSQL i Web Storage, ali i promjenom u trendovima korištenja pojedinih arhitektura, programskih jezika i *frameworka*. Popularnost JavaScripta i modernih *lightweight* arhitektura web servisa poput REST-a doprinijela je širenju JSON (JavaScript Object Notation), odnosno BSON (Binary JSON) formata pohrane koji omogućuju korištenje objekata u svim slojevima aplikacije bez operacija transformacije.

Također je važno napomenuti trend povratka pohrane datoteka potaknut rješavanjem tehničkih problema poput privremene pohrane generiranih datoteka i pristupačnim cijenama takvog oblika pohrane kod popularnih pružatelja cloud usluga. Ekonomski razlozi, odnosno manje cijene rudimentarnijih oblika pohrane kod pružatelja cloud usluga poticatelj su raslojavanja web aplikacija i povećanja kompleksnosti, a Web Storage odnosno lokalna pohrana podataka na strani poslužitelja omogućava korištenje funkcionalnosti aplikacije bez stalne mrežne veze.

3.1. Cloud usluge

"SaaS is a website, PaaS is a framework, IaaS is a virtual machine."

- Tom Limoncelli

Cloud computing je zajednički termin za korištenje računalnih resursa (softvera, platforme, infrastrukture, mreže i dr.) kao udaljene usluge putem računalne mreže - najčešće Interneta. Cloud computing dijelimo u tri osnovna modela prema vrsti resursa: softver kao servis (SaaS), platformu kao servis (PaaS) i infrastrukturu kao servis (IaaS).

IaaS (en. *Infrastructure as a Service*) je model u kojem organizacija koristi vanjske usluge najma računala (najčešće virtualnih računala) i drugih hardverskih resursa koje koristi u obavljanju vlastitog poslovanja. Davatelj usluge zadužen je za čuvanje i održavanje iznajmljenih resursa, a troškovi usluge većinom odražavaju stvarnu uporabu resursa neovisno o vremenskom periodu. IaaS često uključuje i dodatne usluge poput stavljanja na raspolaganje softverskih paketa,

pohrane podataka, unaprijed konfiguriranih virtualnih računala, zakupa IP adresnog prostora (adrese ili raspona/bloka), vatrozida (en. *firewall*) i alata za raspodjelu opterećenja (en. *load balancing*). U najrasprostranjenije IaaS usluge ubrajamo Amazon Elastic Compute Cloud (EC2), Google Compute Engine, Windows Azure Virtual Machines, Oracle Infrastructure as a Service, HP Cloud Compute i Rackspace Cloud Servers.

PaaS (en. *Platform as a Service*) je model u kojem organizacija koristi vanjske usluge koje omogućuju uporabu računalne platforme, odnosno korištenje iznajmljenih alata i biblioteka u izradi, *deploymentu* i *hostingu* vlastitog softvera. Davatelj usluge zadužen je održavanje cjelokupne softverske i hardverske infrastrukture, a često uključuje i dodatne usluge poput *hostinga* baza podataka, alata za timsku suradnju i praćenje grešaka (en. *bug tracking*), te korištenja web i mobilnih servisa poput elektroničke pošte i slanja obavijesti (en. *push notifications*) na mobilne klijente. U najrasprostranjenije PaaS usluge ubrajamo Amazon Web Services Elastic Beanstalk, Google App Engine, Windows Azure Compute, Heroku i VMware Cloud Foundry.

SaaS (en. *Software as a Service*) je model u kojem organizacija koristi softver kao uslugu - davatelj usluge pruža korisnicima pristup aplikacijama i bazama podataka. Davatelj usluge zadužen je za održavanje infrastrukture i platforme na kojoj se aplikacija izvršava. U najrasprostranjenije SaaS usluge ubrajamo Google Apps sa aplikacijama Gmail, Groups, Calendar, Talk, Docs i Sites, Microsoft Office 365 sa aplikacijama Exchange Online, Office Professional Plus, Lync Online, Office Web Apps i SharePoint Online, te OnLive uslugu istoimenog davatelja koja omogućuje igranje računalnih igara smještenih na udaljenim računalima.

Prema Gartnerovoj studiji[3], godišnji trend rasta tržišta javnih cloud usluga iznosi 18%, iako je trend rasta ukupnih IT troškova tek oko 4.4% godišnje, što predlaže veliki trend prelaska sa tradicionalnih na cloud usluge, potaknut brojnim prednostima cloud computinga kao što su:

- skalabilnost resursa i troškova

[3] Gordon et. al., Gartner Worldwide IT Spending Forecast 2Q12, Gartner 2012.

- uštede na ljudskim resursima
- jednostavnija implementacija rješenja kod klijenta
- neovisnost o lokaciji i hardveru
- povećanje pouzdanosti usluge
- alati za analizu i upravljanje
- jednostavnije održavanje
- alati za raspodjelu opterećenja (en. *load balancing*)

Korištenjem cloud usluga često je moguće postići bolju skalabilnost resursa i troškova: promjene u potrebama za računalnim resursima moguće je pratiti brzim povećanjem ili smanjenjem količine zakupljenih resursa, čime se vrijeme odziva na vanjske promjene drastično smanjuje jer ne zahtjeva interne procese nabave ili rashoda vlastitih resursa. Također, troškovi ulaganja u početne resurse novog proizvoda mogu biti znatno manji, a troškovi po jedinici performansi također su znatno manji nego korištenjem vlastitih resursa, što je omogućeno povoljnijom ekonomijom razmjera velikih davatelja usluga. Cloud usluge znatno olakšavaju izvođenje povremenih računski intenzivnih radnji (business intelligence, složena predviđanja i slično) za koje je moguće iznajmiti značajne računalne resurse na kratke vremenske periode.

Cloud usluge često zahtjevaju manje zaposlenih ljudskih resursa u sektoru informacijskih i komunikacijskih tehnologija, budući da davatelji usluge koriste vlastite ljudske resurse za održavanje infrastrukture za pružanje usluge.

Ukoliko organizacija koristi cloud servise u izradi i hostingu softvera koji se zatim prodaje ili nudi kao usluga klijentima, korištenje IaaS i PaaS usluga može skratiti vrijeme i smanjiti kompleksnost implementacije softvera kod klijenta. Razlog tome je korištenje iste, vanjske infrastrukture kao i u procesu razvoja, što znači da troškove održavanja i ljudskih resursa (sistem administratori, administratori baza podataka, mrežni tehničari i drugi) preuzimaju davatelji usluga, što smanjuje kompleksnost i troškove organizaciji koja koristi cloud usluge, te njezinim klijentima. Još jedna prednost te činjenice je moguće smanjenje cijene krajnjeg proizvoda čime proizvođač postaje kompetentniji, a krajnji korisnik ostvaruje uštedu na softveru ili usluzi.

Javne cloud usluge, odnosno cloud usluge dostupne putem Interneta nisu ograničene dostupnosti poput usluga u intranetu organizacije, često su namijenjene korištenju sa vrlo različitih uređaja (stolna računala, laptopi, tableti, pametni telefoni, automobili, pametni domovi, kućanski aparati) i nisu izravno vezani na hosting hardver budući da se virtualna računala izvršavaju na *hypervisor* sloju neovisno o fizičkom hardveru.

Davatelji cloud usluga često jamče dogovorenu minimalnu razinu usluge (SLA en. *Service Level Agreement*), pa tako Amazon EC2 jamči 99.5%, Rackspace Cloud 100%, Google ovisno o korisničkom ugovoru od 95 do 99%, a Microsoft Azure 99.9% jamčene dostupnosti u vremenskom periodu zakupa usluge.

Alati za analizu i upravljanje cloud uslugama omogućavaju uvid u troškove (količine potrošenih resursa trendovi potrošnje), *deployment* kompleksnih aplikacija i konfiguracijskih skripti, automatizaciju sigurnosnih kopija, izdavanje i obnovu digitalnih certifikata i potpisa, provjeru zdravlja (en. *health monitoring*) i automatizaciju procesa.

Cloud pristup pruža centralno mjesto za instalaciju zakrpa i nadogradnji operacijskog sustava, te instalaciju i nadogradnju softvera i sigurnosnih alata. Također, ujedinjeni pristup konfiguracijama (zahvaljujući istovjetnim virtualnim računalima) omogućuje bolju podršku zahvaljujući manjem broju različitih podržanih vrsta uređaja, te pojednostavljenu sigurnosnu reviziju.

Raspodjela opterećenja (en. *load balancing*) je skup metoda koje se koriste za podjelu rada nad većim brojem hardverskih resursa (procesora, mrežnih uređaja, jedinica za pohranu i sl.) s ciljem postizanja optimalnog iskorištenja resursa, smanjenja vremena odziva i izbjegavanja preopterećenja. Brojni cloud servisi nude vlastite alate i servise koji razlčitim metodama (rasporedni algoritmi, eksplicitni prioriteti, *caching* sadržaja, *SSL offload* i drugi) optimalno raspoređuju resurse.

Usprkos brojnim prednostima korištenja cloud servisa i općim trendovima rasta zabilježenih kod svih uobičajenih *cloud computing* modela, on donosi i neke nedostatke:

- očuvanje podataka i intelektualnog vlasništva
- pitanja pravne nadležnosti
- SaaS softver nedostupan po isteku ugovora
- povećanje kompleksnosti arhitekture i sigurnosnih mjera
- nemogućnost prelaska na drugu platformu (en. *vendor lock-in*)

Privatnost podataka i zaštita intelektualnog vlasništva često se ističu kao važni argumenti protiv korištenja cloud usluga. Međutim, Schneier[4] ističe kako informacijske tehnologije neovisno o cloud uslugama uvelike ovise o povjerenju prema raznim entitetima: proizvođačima hardvera, proizvođačima operacijskog sustava i softvera, te pružateljima komunikacijskih usluga. Značajnu razliku u pristupu između korištenja cloud usluga i održavanja vlastite infrastrukture čini povjerenje prema pružatelju usluge u smislu njegove sposobnosti zaštite računalne mreže od vanjskih i unutarnjih napada.

Pitanje pravne nadležnosti, odnosno regulative koja se odnosi na pohranu i primjenu osobnih podataka izvan granica države ili regije, te druge regulative vezane uz računalni softver poput izvoza kriptografskih alata složeno je pitanje koje katkad onemogućava korištenje cloud usluga. Sa ciljem smanjenja spomenutih problema zakonodavstvo se postepeno prilagođava novim uvjetima na tržištu, te se tako otvara mogućnost harmonizacije zakona između zemalja Europske Unije i Sjedinjenih Američkih Država (EU Direktiva 95/46/EC). Također, brojni davatelji cloud usluga već omogućavaju odabir fizičke lokacije, odnosno jasno specificiranje korištenog podatkovnog centra.

Nemogućnost prelaska na drugu platformu (en. *vendor lock-in*) ozbiljan je problem, intenziviran činjenicom da u trenutku pisanja ovog rada ne postoji *de facto* standard vrsta usluga, načina korištenja, te formata pohrane i metoda obrade podataka. Nemogućnost prelaska može biti vezana uz korištenu softversku platformu, način pohrane podataka, dostupnost alata, te način komunikacije i korištenja (en. *consuming*) usluge.

---

[4] Schneier, B., Risks of Cloud Computing, Schneier on Security, http://www.schneier.com/blog/archives/2009/07/risks_of_cloud.html (27.04.2013.)

3.2. Tehnologije upravljanja podacima

Tehologije za upravljanje podacima web aplikacije moguće je kategorizirati prema mjestu i načinu pohrane odnosno korištenja. Prema mjestu, razlikujemo lokalnu pohranu, odnosno svaku pohranu čiji podaci su dostupni nakon prekida mrežne veze, i poslužiteljsku odnosno ne-lokalnu pohranu čijim podacima nije moguće pristupiti u slučaju prekida veze. Prema načinu pohrane, odnosno korištenja, tehnologije za upravljanje podacima dijelimo na:

- datotečnu pohranu (en. *file storage*, *blob storage*, *simple storage*, *binary store*)
- ključnu pohranu (en. *key-value pairs*)
- relacijske baze podataka (en. *relational database*)
- stupičastu pohranu (en. *column-oriented database*)
- pohrana grafova (en. *graph database*)

Datotečna pohrana je najjednostavniji oblik pohrane, čija je najmanja adresabilna jedinica datoteka, odnosno nestrukturirani skup podataka. U kontekstu web aplikacija, navedene su datoteke najčešće dostupne korištenjem HTTP zahtjeva, ili REST API-ja, a popularne cloud usluge omogućuju funkcionalnosti poput zaštite od neautoriziranog pristupa, distribucije sadržaja u CDN (en. *Content Delivery Network*) mrežama, verzioniranje datoteka i grupiranje sadržaja u direktorije i kolekcije, odnosno kontejnere. U popularne cloud usluge koje nude datotečnu pohranu ubrajamo Windows Azure Blob Storage i Amazon Simple Storage Service (S3). Vodeći razlog korištenja datotečne pohrane je *caching*, odnosno mogućnost privremene pohrane generiranih datoteka sa ciljem brže dostupnosti, odnosno boljih performansi prilikom sljedećih posjeta ili posjeta različitih klijenata. Na klijenskoj strani sa ciljem upravljanja datotečnom pohranom iz konteksta web preglednika razvijen je W3C File API standard kojeg već podržavaju svi popularni web preglednici.

Ključna pohrana pruža mehanizme snimanja, pohrane i dohvata podataka na temelju jedinstvenog ključa s kojim je povezan podatak ili set podataka neodređene strukture. Takve ključ-vrijednost (en. *key-value*) strukture uobičajeno ne posjeduju mogućnost formiranja međusobnih veza, izuzev jednostranih pokazivača. Prednost takvih struktura često se izražava kroz performanse, osobito vrijeme dohvata određene vrijednosti, te jednostavnost horizontalnog skaliranja, dok je osnovni

nedostatak nemogućnost korištenja u slučaju strukturiranih sustava kada relacijske sheme nije moguće zamijeniti. Drugi mogući nedostatak odnosi se na performanse snimanja, osobito u slučajevima korištenja složenih indeksa poput izjednačenih stabala (en. *balanced trees*) za pretragu, pri čemu je u slučajevima učestalih dodavanja novih ključeva i vezanih podataka moguć, ovisno o implementaciji, privremeni gubitak performansi u vršnim opterećenjima uslijed dodavanja, ili linearan stabilan pad performansi uslijed narušavanja strukture indeksa. Ključnu pohranu koriste mnogi *caching* mehanizmi (memcached, Redis i slični), te većina rasprostranjenih NoSQL baza podataka, poput BigTable, MongoDB, MemcacheDB, Dynamo, Cassandra, Voldemort i drugih. Veliki nedostatak NoSQL baza podataka jest nedostatak standarda zapisa, upita i migracije, pa je tako ovo područje visoko rizično u smislu vjerojatnosti otežanog prelaska na drugi sustav za upravljanje bazom podataka ili pružatelja cloud usluge. U popularne *cloud* usluge koje nude ključnu pohranu ubrajamo Amazon DynamoDB, Amazon SimpleDB, Google App Engine Datastore, Windows Azure Table Storage i brojne druge.

Na klijentskoj strani, dvije relevantne tehnologije pružaju mehanizme za snimanje ključ-vrijednost struktura: kolačići (en. *cookies*) i HTML5 Web Storage. Kolačići su iznimno rasprostranjena tehnologija koju koristi gotovo svaka web aplikacija, međutim imaju svoja ograničenja: najveća dozvoljena veličina kolačića je 4kb, i svi kolačići koje web preglednik posjeduje za određenu domenu prenose se sa klijentskog preglednika prilikom svakog HTTP(S) zahtjeva. Izvorno dio HTML5 specifikacije, Web Storage tehnologija dozvoljava pohranu 5MB podataka po domeni i pruža bogati JavaScript API za manipulacijom podacima. Za razliku od kolačića, Web Storage podaci ne prenose se prema poslužitelju sa svakim zahtjevom. Web Storage podržavaju svi popularni web preglednici u trenutno stabilnim dostupnim inačicama.

Relacijske baze podataka usprkos svojoj starosti ostaju vodeći izbor u većini web aplikacija zahvaljujući brojnim prednostima, kao što su ACID svojstva, raširenи i rasprostranjeni protokoli, općeprihvaćeni SQL upitni jezik, veliki broj besplatnih sustava za upravljanje bazama podataka, mnoštvo literature, lako dostupna podrška i nativna podržanost na većini platformi, alata za razvoj i operacijskih sustava, osobito u pogledu jezika, protokola i drivera. U nedostatke sustava za upravljanje relacijskim bazama podataka, i relacijskog modela uopće ubrajamo umanjenje performansi s ciljem očuvanja konzistentnosti, relativno stalnu odnosno rigidnu shemu i

strukturu, otežano horizontalno skaliranje, te u nekim slučajevima povećane cijene korištenja povezanih cloud usluga. Velika prednost relacijskih baza podataka svakako je i mogućnost relativno jednostavne migracije sustava sa jedne na drugu relacijsku bazu zahvaljujući sličnosti podatkovnih tipova, pristupa, upitnog jezika i relativno malih zahvata u programskom kodu podatkovnog sloja. Zahvaljujući njihovoj popularnosti, svi vodeći pružatelji cloud usluga u ponudi sadrže barem jedan sustav za upravljanje relacijskim bazama podatka: Heroku (MySQL, PostgreSQL), Windows Azure (SQL Azure), Amazon Relational Database Service (MySQL), Google Cloud SQL (MySQL) i Oracle Database Cloud Services (Oracle SQL). Na klijenskoj strani podrška za relacijske baze podataka uvedena je putem W3C Web SQL Database standarda, međutim navedeni standard podržavaju tek Google Chrome i Apple Safari preglednici, te je predviđeno njegovo ukidanje. Kao zamjenu uvodi se W3C Indexed Database API koji omogućuje pohranu ključ-vrijednost parova i hijerarhijske veze između pojedinih objekata. Mozilla Firefox, Internet Explorer 10 i Google Chrome podržavaju novi nacrt standarda u trenutnim stabilnim verzijama.

Stupičasta pohrana (en. *column-oriented storage*) princip je pohrane podataka u kojem su podaci pohranjeni kao skupine stupaca, za razliku od relacijskog modela u kojem su podaci pohranjeni kao skupine redaka. Takva organizacija podataka ima arhitekturalne prednosti veće brzinu obrade skupova istovrsnih podataka, pa se iz tog razloga koriste u skladištima podataka (en. *data warehouses*). Stupičasta pohrana ne odnosi se samo na način raspodjele podataka unutar baze podataka već i na metode obrade, pa tako samom promjenom sustava za upravljanje bazom podataka nije moguće iskoristiti prednosti takve reorganizacije arhitekture. Većinsku uštedu vremena u takvoj reorganizaciji čini trajanje vremena potrage (en. *seek time*), čija je hardverski najznačajnija komponenta kod rotacijskih diskova moment kretnje ploče i pokret glave. Promjenom tehnologija fizičke stalne pohrane sa rotacijskih diskova na *solid state* diskove, vremena potrage značajno se poboljšavaju pa tako u budućnosti možemo očekivati manji interes za stupičastom pohranom potaknut nestajanjem hardverskih limitacija. Unatoč mogućim budućim trendovima, osnovne razlike između stupčastih i retkovno-orijentiranih baza podataka su:

- izmjene koje se odnose na veliki broj istovrsnih atributa izvršavaju se brže u stupičastim bazama podataka

- izmjene svih istovrsnih atributa mnogostruko su brže u stupičastim bazama podataka jer nije potreban prolazak kroz ostale atribute retka
- retkovno dohvaćanje podataka, odnosno dohvaćanje cjelokupnih entiteta sa svim njegovim atributima mnogostruko je brže u retkovnim bazama podataka
- zapisivanje i brisanje novog retka mnogostruko je brže u retkovnim bazama podataka

Ne postoji raširena klijentska tehnologija niti pripadajući standard stupičaste pohrane, budući da su prednosti takve arhitekture uočljive tek kod velikih setova podataka kakve nije praktično niti smisleno smještati u web preglednik.

Pohrana grafova odnosno graf-struktura u specijalizirane baze podataka tradicionalno je područje zanimanja računalnih znanosti, koje se posljednjih godina dovodi u okvire programskog inženjerstva odnosno primjenjenog računarstva zahvaljujući prvenstveno potrebi za upravljanjem prostornim, odnosno geografskim podacima. U odnosu na relacijske baze podataka, graf baze uobičajeno omogućuju bržu pretragu usko povezanih entiteta, omogućuju lakše horizontalno i vertikalno skaliranje i vrlo jednostavnu promjenu interne strukture. Relacijske baze omogućuju višekratno brže izvođenje operacija nad setovima različitih ali istovrsnih entiteta. U popularne graf baze ubrajamo AllegroGraph, Neo4j, FlockDB, InfiniteGraph i GraphDB.

4. Poslovni sloj i komunikacija

Posljednjih godina, sljedeći su faktori uvelike utjecali na arhitekturu i mogućnosti poslovnog sloja i komunikacije web aplikacija: evolucija web frameworka i alata, pojava JavaScript platformi na poslužiteljskoj strani, napredak performansi JavaScripta i raširenosti bogatih JavaScript biblioteka na klijentskoj strani, standardizacija novih HTML5 srodnih API-ja, te pojava biblioteka i protokola za *real-time web* komunikaciju.

U nastavku se opisuje ASP.NET MVC4 web framework kojeg autor smatra reprezentativnim predstavnikom modernih web frameworka sa mnoštvom ugrađene funkcionalnosti koja omogućuje *high-level*, rapidan razvoj cjelokupne web aplikacije. Zatim slijedi SignalR, paket za *real-time web* komunikaciju na .NET platformi i predstavnik novih *real-time web* biblioteka koje koriste prednosti HTML5 WebSockets tehnologije. U sljedećem dijelu, opisuje se node.js tehnologija, predstavnik nove skupine *lightweight web frameworka* sa težištem postavljenim na performanse, modularnost i korištenje jednog jezika (JavaScript) i jednog transportnog formata objekata (JSON) u svim slojevima aplikacije. Od brojnih popularnih *front-end frameworka* odabran je Knockout.js, MVVM framework namjenjen lakšem usklađivanju korisničkog sučelja i podataka sustava u stvarnom vremenu, te jasnijom separacijom podataka od prikaza na klijentskoj strani.

4.1. ASP.NET MVC4

ASP.NET MVC je web framework otvorenog koda namjenjen izradi web aplikacija Model-View-Controller arhitekture na .NET platformi. Usprkos kasnoj pojavi (2009.) u odnosu na brojne rasprostranjene web frameworke na drugim platformama poput Zend Framework (2006.), Google Web Toolkit (2006.), Django (2005.) i Ruby on Rails (2006.), ovogodišnji ASP.NET MVC4 sa srodnim tehnologijama posjeduje sve važne karakteristike modernog web frameworka:

- podrška za pisanje REST servisa putem Web API paketa
- podrška za *real-time web* funcionalnosti putem SignalR paketa

- mogućnost pisanja *code-first* (generiranja baze podataka iz poslovnih objekata) ili *data-first* (generiranja ORM klasa iz baze podataka) web aplikacija
- moderan ORM sustav (Entity Framework) sa ugrađenom ili dostupnom podrškom za SQL Server, SQL Azure, Oracle, MySQL, PostgreSQL, SQLite i druge popularne sustave za upravljanje bazama podataka
- podrška za Test-Driven Development i cjelomično generiranje testnog okruženja
- podrška za Dependency Injection (DI) i Inversion of Control (IoC) komponentne modele
- mehanizmi za dinamičko generiranje URL ruta za jednostavniju optimizaciju za tražilice (en. *search engine optimization*) i adresiranje REST servisa
- *templating* jezik za dinamičko generiranje pogleda i parcijalnih pogleda sa mogućnošću *strongly-typed* modela pogleda
- integracija *scaffolding* mogućnosti generiranja stranica i formi iz poslovnih objekata
- podrška za samostalnu autentikaciju i autorizaciju, Windows domenske identitete, te članstvo i role uz generiranje odgovarajućih podatkovnih struktura
- ugrađeni *caching* mehanizmi i mehanizmi za održavanje korisničkih sesija
- Visual Studio podrška za *debugging*, te JavaScript, CoffeeScript, TypeScript, LESS i SASS skriptne i stilske jezike
- integralno upravljanje paketima i ekstenzijama putem NuGet Visual Studio ekstenzije

4.2. Real-time web

Real-time web tehnologije omogućuju dostavu sadržaja sa poslužitelja na klijente u trenutku kada sadržaj postane postupan. Iako na prvi pogled trivijalan i odavno riješen, ovaj problem zapravo je tek prividno riješen na većini rasprostranjeih web sjedišta i aplikacija koristeći česte AJAX klijentske zahtjeve za novim sadržajem (en. AJAX *polling, successive AJAX requests*). Takav pristup, iako do sada neizbježan predstavlja veliko opterećenje za poslužitelje i mrežne resurse zbog vrlo čestih uzastopnih zahtjeva, dok smanjenje frekvencije zahtjeva smanjuje vrijeme odziva, odnosno reakcije na nove sadržaje: stvarajući uvijek kompromis između količine alociranih resursa i spremnosti odricanja na brzi vremenski odziv. AJAX se oslanja na

tehnologiju XMLHttpRequest (XHR), implementiranu još 1995. godine u pregledniku Netscape Navigator.

Tehnika zvana *long polling* posljednjih godina postepeno zamjenjuje *AJAX polling*, međutim i dalje se radi o *half-duplex* tehnologiji. Osnovna razlika AJAX pollinga i long pollinga jest trenutak slanja odgovora sa poslužiteljske strane: ukoliko poslužitelj ne posjeduje novi sadržaj za klijenta, umjesto slanja praznog odgovora poslužitelj zadržava TCP zahtjev do trenutka dostupnosti novog sadržaja. Nakon što sadržaj postane dostupan (i ukoliko ne istekne dogovoreni *timeout* period) odgovor biva poslan klijentu. Klijent će obično odmah potom uputiti novi korisnički zahtjev koji će biti zadržan na poslužiteljskoj strani do trenutka kada je cjelovit odgovor moguć.

Dvije naprednije tehnologije, Forever Frame i Server Sent Events možemo nazivati istinskim *real-time web*, odnosno *push* tehnologijama budući da omogućuju stvarnu *full duplex* komunikaciju. Forever Frame tehnika upotrebljuje funkcionalnost uvedenu već u W3C HTML v1.1 standardu namijenjenu inkrementalnom slanju velikih dokumenata, te omogućuje nisku latenciju jer izbjegava stvaranje i zbrinjavanje velikog broja pojedinačnih HTTP i TCP veza koristeći jednu stalnu vezu za prijenos. Nedostatci ove tehnike su nemogućnost pouzdanog upravljanja iznimkama, nemogućnost praćenja stanja zahtjeva i nedostatna podrška od strane proizvođača web preglednika. Druga značajnija *push* tehnologija naziva se Server-Side Events, i omogućuje slanje događaja sa poslužitelja prema preglednicima u obliku DOM (en. *Document Object Model*) događaja. Najveći nedostatak ove tehnologije jest manjak podrške u Microsoftovim preglednicima. Trendovi rasprostranjenosti pojedinih preglednika u posljednjim godinama sugeriraju kako navedeni problem poprima sve manju važnost, međutim sljedeća tehnologija pokazuje najveći potencijal da postane *de facto* standard za *real-time web* funkcionalnost u nadolazećim godinama.

WebSockets i pripadajući WebSockets API dio su tekućeg HTML5 radnog standarda za komunikaciju klijent-poslužitelj, koji pruža novi model uspostave veze i obostranog slanja podataka. HTTP protokol zahtjeva uspostavu veze rukovanjem (en. *handshake*), najčešće na portovima 80, 8080 i 443 u slučaju TLS/HTTPS. Nakon uspostave, paketi se razmjenjuju putem

istih portova i veza između klijenta i poslužitelja se prekida. WebSockets protokol također vrši proces rukovanja na uobičajenim portovima, međutim nakon uspostave veze klijent i poslužitelj dogovorom premještaju komunikacijski kanal na neki viši, posvećeni port. Time uobičajeni portovi ostaju dostupni za daljnju komunikaciju sa drugim stranama, a prividno neprekidna WebSockets veza na dogovorenom visokom portu ostaje otvorena do daljnjega. WebSockets standard predviđa uobičajeni format poruke, te API-je sa klijentske i poslužiteljske strane kojima se upravlja događajima poput primanja poruke, te prekida i ponovne uspostave veze.

Microsoftov SignalR je poslužiteljska i klijentska biblioteka namjenjena jednostavnoj i brzoj implementaciji *real-time web* funkcionalnosti u .NET web sjedišta i aplikacije. SignalR se oslanja na HTML5 WebSockets API ukoliko ga klijent i poslužitelj podržavaju, ali ima ugrađenu podršku i za upotrebu ranijih tehnologija (Forever Frame, Serve-Side Events, Long Polling, AJAX) dostupnih na starijim preglednicima, bez potrebe za promjenom programske logike. Poslužiteljske SignalR biblioteke namjenjene su upotrebi u .NET Framework aplikacijama, dok su klijentske datoteke napisane za .NET, ali i JavaScript klijente, odnosno pretežito web preglednike. SignalR je uključen u Microsoft ASP.NET and Web Tools 2012.2 dodatak za Visual Studio 2012 razvojnu okolinu u veljači 2013. godine, a podržan je u svim modernim preglednicima (IE6+), međutim punu WebSockets funkcionalnost moguće je postići sa Internet Explorer 10, Mozilla Firefox 11+, Google Chrome 16+ i Apple Safari 6+, dok testovi autora rada ukazuju i na punu funkcionalnost na Apple iPhone i iPad mobilnim uređajima (iOS 6), te Google Android 4 i Windows Phone 8 pametnim telefonima i tabletima.

SignalR pruža dva različita modela za *real-time web* komunikaciju: Persistent Connection i Hubs. Persistent Connections predstavljaju ekonomičniji pristup u pogledu alokacije resursa, a omogućuju slanje JSON tipova od poslužitelja prema klijentu i string tipova od klijenta prema poslužitelju. Persistent Connections i Hubs oboje omogućuju grupiranje veza, pa tako poslužitelj može obavijestiti pojedinog klijenta, grupu klijenata ili sve trenutne klijente o nastalom događaju. Hubs u funkcionalnom ali i objektno-orijentiranom smislu predstavljaju nasljednika Persistent Connections sa nešto većom potrošnjom resursa i značajno proširenom funkcionalnošću. Hubs omogućuju spajanje modela (en. *model binding*), odnosno slanje JSON objekata sa klijentske strane uz automatsku serijalizaciju pri pristizanju na poslužitelj u korisnički definiranu klasu

modela. Također, Hubs omogućuje pozive udaljenih metoda (en. *remote procedure calls*), odnosno mogućnost pozivanja klijentskih metoda sa poslužiteljske strane, i poslužiteljskih metoda sa klijentske strane. Za potrebe ovog rada koristiti ćemo isključivo napredniji Hubs model.

Stvaranje vlastite Hub funkcionalnosti započinje naslijeđivanjem Hub klase. Sve javne metode korisničke klase implicitno postaju javno dostupne pokretanjem aplikacije, uz prethodno konfiguriranje koje se sastoji od uključivanja klijentske biblioteke u sve web stranice koje koriste SignalR funkcionalnost i prijavu ASP.NET MVC rute putem koje će se vršiti komunikacija između svih Hub nasljednika i klijenata.

```
public class OrangeFish : Hub {
    public string JavnoDostupnaMetoda(string arg1, MojaKlasa arg2)
    {...}
}
```

SignalR Hubs posjeduje nekoliko metoda namjenjenih rukovanju događajima uspostave veze, prekida veze, te ponovne uspostave:

```
public override System.Threading.Tasks.Task OnConnected() {
    return base.OnConnected();
}
public override System.Threading.Tasks.Task OnReconnected() {
    return base.OnReconnected();
}
public override System.Threading.Tasks.Task OnDisconnected() {
    return base.OnDisconnected();
}
```

Svaki Hubs nasljednik ima pristup Context objektu, kojim se prenosi cjelokupan klijentski kontekst, odnosno podaci o vezi, autentifikaciji, pregledniku, GET parametrima, formi i slično.

```
/* identifikator veze, npr. „3bbcf377-74d5-47cd-90e7-4e11fafa3ec4“ */
var _id = Context.ConnectionId;

/* user-agent string preglednika, npr. „Mozilla/5.0 (Windows NT 6.1)“ */
var _user_agent = Context.Headers[„User-Agent“];
```

```
/* korišteni transport, npr. „serverSentEvents“ */
var _transport = Context.QueryString[„transport“];

/* podaci iz poslane forme */
var _form = Context.Request.Form[„data“];

/* stanje autentifikacije korisnika */
var _auth = Context.User.Identity.IsAuthenticated;
```

Putem Groups objekta upravlja se grupama, odnosno kolekcijama identifikatora veza. Clients objekt koristimo za udaljene pozive funkcija (en. *remote procedure call)* na različitim klijentima.

```
/* dodavanje veze (prema identifikatoru) grupi */
Groups.Add(Context.ConnectionId, „NazivGrupe“);

/* brisanje veze iz grupe */
var _druga_veza = „4aaff275-26d4-11bc-00a1-3c23abcc5fc8“;
Groups.Remove(_druga_veza, „NazivGrupe“);

/* pozivanje funkcije na svim klijentima */
Clients.All.funkcijaNaKlijentu(„argument“);

/* poziv prema svim klijentima, osim Context.ConnectionId */
Clients.Others.funkcijaNaKlijentu(„argument“);

/* poziv prema svima, osim izuzetka ili liste izuzetaka */
Clients.AllExcept(Context.ConnectionId).funkcijaNaKlijentu(„argument“);

/* poziv isključivo prema pozivatelju (Caller = Context.ConnectionId) */
Clients.Caller.funkcijaNaKlijentu(„argument“);

/* poziv prema grupi */
Clients.Group(„NazivGrupe“).funkcijaNaKlijentu(„argument“);

/* poziv prema grupi, osim izuzetka ili liste izuzetaka */
var _iskljuceni = new string[] { Context.ConnectionId, „4aaff275...“ };
Clients.Group(„NazivGrupe“, _iskljuceni).funkcijaNaKlijentu(„argument“);

/* poziv prema pojedinom klijentu prema identifikatoru */
Clients.Client(_id).funkcijaNaKlijentu(„argument“);
```

Iako važna iz perspektive performansi, WebSockets tehnologija ne mijenja komunikacijski model s aspekta korisnika, odnosno činjenicu da se komunikacija uvijek odvija između poslužitelja i

klijenta, ili više klijenata posredstvom poslužitelja. Nadolazeća generacija *real-time web* tehnologija omogućiti će uspostavu izravnog komunikacijskog kanala između dva klijenta odnosno web preglednika. Nacrt W3 WebRTC standarda predlaže načine komunikacije koji uključuju glasovne razgovore odnosno razmjenu zvučnog signala, video komunikaciju i razmjenu datoteka između klijenata.

Navedena tehnologija omogućiti će tako prvi novi model komunikacije web preglednika od implementacije asinkronih zahtjeva (AJAX), i vjerojatno započeti novu revoluciju inovativnih web usluga i načina korištenja mrežno povezanih uređaja. Novim komunikacijskim modelom web preglednici osiguravaju svoju dominantnu poziciju središta umreženog života i postaju konkurent nativnim komunikacijskim klijentima, te alatima za razmjenu digitalnog sadržaja. Jednostavno dostupna razmjena datoteka koristeći kriptirani komunikacijski kanal zasigurno će imati implikacije i na zakonodavstvo i tehnička sredstva kontrole digitalnog sadržaja. Trenutno dostupne stabilne verzije preglednika Mozilla Firefox i Google Chrome implementiraju većinu mogućnosti nacrta WebRTC standarda.

4.3. JavaScript na poslužitelju i klijentu

Iz perspektive programiranja, zasigurno najznačajnija promjena na području web aplikacija u posljednjih nekoliko godina jest mogućnost korištenja jednog programskog jezika – JavaScripta na poslužiteljskoj i klijentskoj strani, odnosno svim programabilnim slojevima web aplikacije. U skladu sa dobivenom važnošću, JavaScript kao standard ali i živući programski jezik mijenja se u smjeru snažnije objektne orijentiranosti, modularnosti, veće učinkovitosti i značajnije kontrole kvalitete i stila kodiranja. Promjene su također izražene i pojavom iznimno velikog broja novih JavaScript biblioteka na sjedištima poput GitHuba, te alata za *workflow* optimizaciju i automatizaciju, statičku analizu, transpilaciju, minifikaciju, dijeljenje koda i testiranje.

Kašnjenjem ECMAScript 6, odnosno novog JavaScript standarda koji dovodi brojne funkcionalnosti etabliranih objektno orijentiranih i funkcijskih jezika (klase, moduli, iteratori, *arrow* funkcije, binarni tipovi, *generator expressions, dynamic proxies,* kolekcije...) javlja se potreba za stvaranjem nad-jezika, odnosno jezika koji navedene i druge funkcionalnosti mogu

procesom transpilacije dovesti u JavaScript izvršno okruženje. Najpopularniji jezici namjenjeni transpilaciji u JavaScript su CoffeScript – inspiriran Rubyjem, Pythonom i Haskellom te trenutno deseti najpopularniji jezik na GitHubu i TypeScript – razvijen u Microsoftu i inspiriran jezikom C#.

Najpopularnije moderne JavaScript pakete možemo smjestiti u aplikacijske *frameworke* (AngularJS, Backbone.js, Ember.js), *model binding* i *templating* biblioteke (KnockoutJS, Handlebars.js, mustache.js, Dust.js), DOM manipulaciju, povezanost sa poslužiteljem i uslugama, vizualizaciju i prezentaciju, te biblioteke i *widgete* korisničkog sučelja.

S ciljem unaprijeđenja kvalitete JavaScript koda stvoreni su brojni alati za statičku analizu, od kojih su najpopularniji JSLint[5] i njegov konfigurabilniji *fork* JSHint, te alati za analizu DOM modela web aplikacija, kao što je DOM Monster. Značajna je i pojava online editora i alata za pisanje JavaScript aplikacija (Cloud9 IDE, ShiftEdit, codeanywhere), te servisa za prototipiziranje i dijeljenje *snippeta* (jsFiddle, Tinkerbin, Pastebin).

Rastom broja *front-end* biblioteka uključenih u pojedinu web aplikaciju, javljaju se problemi koji su donedavno bili svojstveni *back-end* razvoju: *workflow* optimizacija, upravljanje paketima, automatizacija *build* procesa, *lazy loading* resursa, te problemi vezani uz prijenos podataka i performanse preglednika, poput minifikacije i kombiniranja biblioteka te uranjene kompilacije predložaka. Kako bi se poboljšale performanse učitavanja web aplikacija, upotrebljavaju su biblioteke poput RequireJS koje omogućuju asinhrono učitavanje resursa poput datoteka i modula odnosno biblioteka u najkasnijem trenutku, smanjujući time broj resursa potreban za inicijalno učitavanje aplikacije. Upravljanje paketima omogućeno je putem projekata poput Bowera – sustava za upravljanje paketima cjelokupnog *front-enda* koji omogućuje definicije zavisnosti paketa, koristi Git za verzioniranje te posjeduje API za konzumiranje paketa. Brojni procesi *front-end* razvoja poput JavaScript i CSS transpilacije, uranjene kompilacije predložaka, statičke analize, izvršavanja *unit* testova te kombiniranja i minifikacije paketa mogu se automatizirati

[5] http://www.jslint.com/lint.html (04.05.2013.)

koristeći GRUNT i slične projekte, dok se cjelokupni *workflow* može provoditi alatima poput Yeomana ili vlastitog *forka* projekta.

4.3.1. Node.js

Node.js je softverska platforma bazirana na Google V8 *JavaScript engine*-u namjenjena pisanju *event-driven* web aplikacija, sa težištem na postizanju visokih performansi, visokoj skalabilnosti i malom *overhead*-u. Node.js aplikacija ujedno je i web poslužitelj (odnosno, node.js je istovremeno biblioteka i *runtime* okruženje), omogućujući tako potpunu kontrolu nad načinom rada poslužiteljske programske logike.

Za razliku od tradicionalnih web poslužitelja, node.js ne blokira nove zahtjeve dok izvršava postojeći, već stalno prima pojedine zahtjeve i asinkrono odgovara na njih redoslijedom izvršavanja. Takav pristup nazivamo ne-blokirajućim ili *event-driven* ulazom/izlazom.

U nastavku slijedi primjer jednostavne node.js aplikacije koja posjeduje HTTP poslužitelj, logiku upravljanja URL rutama, dostavu statičkog sadržaja sa datotečnog sustava, te upravljanje greškama:

```
var http = require('http'); //biblioteka HTTP poslužitelja
var url = require('url');   //url parsiranje
var fs = require('fs');     //pristup datotečnom sustavu

http.createServer(function (req, res) { //proslijeđivanje anonimne callback
metode
    var url_parts = url.parse(req.url);

    //upravljanje rutom
    var reg_css = new RegExp('^(/css/)');
    if (reg_css.exec(url_parts.pathname) != null) {
        handleDeliver('.' + url_parts.pathname, 'text/css')
    } else {
        switch (url_parts.pathname) {
            case '/':
                handleDeliver('./index.html', 'text/html')
                break;
            case '/about':
                handleDeliver('./about.html', 'text/html')
                break;
            default:
                handleNotFound(url_parts.pathname, req, res);
        }
    }
    return;
```

```
    //dostava sadržaja (status code 200)
    function handleDeliver(path, mime) {
        console.log('200 on ' + path + '(' + mime + ')');
        fs.readFile(path, function (error, content) {
            if (error) {
                handleError(error);
            } else {
                res.writeHead(200, { 'Content-Type': mime });
                res.end(content, 'utf-8');
            }
        });
    }

    //resurs nije pronađen (status code 404)
    function handleNotFound(url, req, res) {
        console.log('404 on ' + url);
        res.writeHead(404, { 'Content-Type': 'text/html' });
        res.write("<h1>404</h1>");
        res.end("The page '" + url + "' could not be found :/");

    }

    //greška (status code 500)
    function handleError(error) {
        console.log('500 on ' + error);
        res.writeHead(500);
        res.end();
    }

}).listen(8888);    //poslužitelj aktivan na portu 8888
console.log('Server running at http://localhost:8888/');
```

Važno je napomenuti kako je node.js iznimno *lightweight*, odnosno posjeduje malo ugrađene funkcionalnosti, međutim većinu klasične *web framework* funkcionalnosti moguće je implementirati putem raznih *third-party* biblioteka (modula), kao što su node_redis (Redis klijent), cradle (CouchDB biblioteka), note-static (posluživanje statičkog sadržaja), async (asinkrono programiranje) i jade (tempating engine). Node.js također posjeduje vlastiti *package manager* i driver za MongoDB sustav za upravljanje NoSQL bazom podataka.

4.3.2. Knockout.js

Kao primjer modernog klijentskog JavaScript *frameworka* navodimo Knockout.js: biblioteku koja implementira MVVM arhitekturu na klijentskoj strani, odnosno pruža jasnu separaciju komuniciranih (transportnih) podataka, podataka za prikaz (odnosno modela prikaza), programske logike prikazivanja i samog prikaza. Knockout posjeduje sljedeće funkcionalnosti:

- praćenje zavisnosti (en. *dependency tracking*) – automatsko osvježavanje prikaza promjenom podatkovnog modela, ili obrnuto: osvježavanje podatkovnog modela i prijenos izmjena na poslužiteljsku stranu izmjenom povezanih komponenti sučelja
- deklarativno povezivanje (en. *declarative binding*) – deklaraciju podatkovnih veza u markupu (HTML strukturi) sloja prikaza
- *templating* – pisanje HTML predložaka i njihovo kompiliranje i renderiranje potaknuto događajima
- *computed observables* – pisanje vlastite transformacijske logike za pretvaranje domenskih podataka u odgovarajuće podatke prikaza
- ulančavanje zavisnosti (en. *dependency chaining*) – stvaranje podataka prikaza ovisnih ili kalkuliranih o drugim podacima prikaza u neograničenim lancima

U nastavku slijedi prikaz osnovne Knockout.js funkctionalnosti:

```
<html>
<head>
<style type="text/css">
    .kockica {display: inline-block; width: 8px; height: 8px;}
</style>
<script src="jquery-2.0.0.min.js"></script>
<script src="knockout-2.2.1.js"></script>

<script type="text/javascript">
    $(function () {
        function MojViewModel() {
            var self = this;
            /*dodijeljena vrijednost*/
            self.ime = "Nikola";
            /*observable izmjenjiva promatrana vrijednost*/
            self.prezime = ko.observable("Tesla");
            /*kalkulirana rijednost*/
```

```
            self.punoIme = ko.computed(function () {
                return self.ime + " " + self.prezime();
            }, self);
            self.bitcoin = ko.observable();
            /*javno dostupna metoda za izmjenu vrijednosti*/
            self.osvjeziBitcoin = function (val) {
                self.bitcoin(Math.round(Math.random() * 10000) / 100 + '
USD');
            };
            /*izmjenjivo polje objekata*/
            self.boje = [
                { ime: "Crvena", vrijednost: "f00" },
                { ime: "Zelena", vrijednost: "0f0" },
                { ime: "Plava", vrijednost: "00f" }
            ];
        }
        var vm = new MojViewModel();
        /*primjena instanciranog viewmodela*/
        ko.applyBindings(vm);
        /*inicijalna vrijednost*/
        vm.bitcoin('Nepoznato');
    });
</script>
</head>

<body>
    <!-- data-bind tipa text je jednostrano izmjenjiv-->
    <p>Ime: <strong data-bind="text: ime"></strong></p>
    <!-- data-bind tipa value je obostrano izmjenjiv -->
    <p>Prezime: <input data-bind="value: prezime" /></p>
    <p>Puno ime: <strong data-bind="text: punoIme"></strong></p>
     <!-- poziv metode viewmodela na događaj (klik) -->
    <button data-bind="click: osvjeziBitcoin">Osvježi BTC </button>
    <p>Bitcoin: <strong data-bind="text: bitcoin"></strong></p>
    <ul data-bind="foreach: boje"> <!-- templating -->
        <li>
            <span data-bind="text: ime"></span>
            <div data-bind="style: { background: '#' + vrijednost }"
class="kockica"></div>
        </li>
    </ul>
</body>
</html>
```

Knockout.js i slične biblioteke posebno su korisne u dva scenarija: web aplikacijama napisanim u jednostraničnoj (en. *single-page*) arhitekturi i aplikacijama koje implementiraju *real-time web* funkcionalnosti. Kod jednostraničnih aplikacija, funkcionalne cjeline aplikacije dijelovi su iste klijentske okoline, odnosno ne zahtjevaju sinkrona učitavanja stranice ili njezinih dijelova. Poslovni podaci bivaju dohvaćeni u JSON i sličnim formatima asinkronim tehnikama (AJAX, WebSockets, WebRTC), dok klijentski *framework* poput Knockout.js-a osigurava obostranu sinkronizaciju korisničkog sučelja i pozadinskog poslovnog sustava. Moderne *push* tehnologije poput HTML5 WebSockets osiguravaju događajno-uvjetovanu pravovremenu reakciju na poslužiteljsku ili klijentsku izmjenu, najčešće koristeći neku od *real-time web* biblioteka poput SignalR na .NET ili socket.io na node.js platformi, pri čemu Knockout.js osigurava sinkronizaciju sa korisničkim sučeljem, odnosno DOM strukturom i stilskim (CSS) značajkama.

5. Prezentacijski sloj

5.1. HTML5

"New open standards created in the mobile era, such as HTML5, will win."
- Steve Jobs

U odnosu na ranije verzije HyperText Markup Language specifikacija (kao što su W3C HTML 4.01, W3C XHTML 1.1), HTML5 donosi veliki broj važnih noviteta, kao što su novi semantički elementi i atributi, poboljšanja web formi, sučelja za web aplikacije (API en. *Application Programming Interface*), proširene mogućnosti iscrtavanja (Canvas, Scalable Vector Graphics) i nativni multimedijski elementi poput zvuka i video sadržaja.

Nažalost, u trenutku pisanja ovog rada nije moguće reći da je postignut konsenzus u definiranju točnog skupa tehnologija koje definiraju HTML5. Također, neovisno o definiranju samog standarda, različiti preglednici ne podržavaju iste HTML5 tehnologije, niti postoji dogovoreni redoslijed kojim bi se osigurala većinska kompatibilnost određenih mogućnosti, ili vremenski rokovi implementacije.

Iako je W3C, odnosno World Wide Web Consortium vodeća međunarodna organizacija na području web standarda, njezin utjecaj već dugi niz godina slabi zbog vlastite tromosti u uspostavljanju novih web standarda koji bi pratili potrebe razvoja modernih web aplikacija, kao što su jednostavnija asinhrona komunikacija, standardizirana alternativa XML-u u razmjeni podataka, semantika koju tražilice mogu kvalitetnije interpretirati, dogovoreni otvoreni formati za multimediju (zvuk, video, tipografija), manipulacija grafikom u pregledniku, podrška za nove hardverske mogućnosti mobilnih uređaja (akcelerometar, jačina svjetla, jačina pritiska) i slično.

Iz navedenih razloga, 2004. godine grupa programera iz Applea, Mozille i Opere osniva WHATWG, odnosno Web Hypertext Application Technology Working Group čija je osnovna zadaća pravovremeno standardiziranje nove HTML specifikacije i njezinu implementaciju u vodećim preglednicima. Prijedlog WHATWG grupe jest stalna evolucija HTML specifikacije prema nadolazećim potrebama i bez jasne distinkcije između pojedinih verzija, odnosno

postojanje fluidne specifikacije prema kojoj bi se nadograđivala programska implementacija popularnih preglednika. 2007. godine W3C preuzima tekuću WHATWG specifikaciju na temelju koje stvara početnu točku W3C HTML5 standarda, odnosno First Public Working Draft.

Ukoliko HTML5 definiramo kao skup svih nadogradnji i promjena postojećih HTML 4.01 i XHTML 1.1 specifikacija koje su vodeći proizvođači preglednika (Google, Mozilla, Microsoft i Apple) implementirali u posljednje verzije svojih preglednika ili većinski najavili njihovu implementaciju, možemo zaključiti da HTML5 posjeduje sljedeća svojstva:

- novi semantički elementi i atributi koji odražavaju stvarne potrebe web stranica i aplikacija
- nova sintaksa za umetanje multimedijalnih sadržaja poput zvuka i video materijala
- uklanjanje prezentacijskih elemenata iz standarda
- mogućnost prikaza vektorske grafike (SVG) i matematičke notacije (MathML)
- nova sučelja za interakciju sa stranicom putem skriptnih jezika:
- canvas element za 2D iscrtavanje
- upravljanje reprodukcijom zvučnih i video materijala
- offline funkcionalnost web aplikacija (izvođenje i pohrana)
- uređivanje dokumenata
- drag-and-drop
- među-domenska komunikacija
- upravljanje poviješću preglednika
- upravljanje MIME tipovima
- umetanje Microdata semantičkih oznaka za poboljšavanje pretraživanja
- pohrana strukturiranih podataka (Web Storage, Indexed Database API, Web SQL Database)
- geolokacija
- upravljanje datotekama
- otkrivanje značajki (en. *feature detection*)

Raznovrsnost web preglednika na različitim klijentima i platformama upućuje na potrebu otkrivanja podrške za pojedinim HTML5 mogućnostima na klijentskoj strani, kako bi se web stranica odnosno aplikacija prilagodila konkretnim mogućnostima na pojedinim klijentima. S tim ciljem, dogovorene su sljedeće metode za otkrivanje svojstava (en. feature detection) koristeći DOM (en. Document Object Model)

1. provjera postojanja atributa na globalnom objektu (window, navigator i sl.)
2. provjera postojanja atributa na stvorenom elementu (<canvas> i sl.)
3. provjera postojanja metode na stvorenom elementu (<video> i sl.)
4. provjera zadržavanja vrijednosti atributa (<input> tipovi)

Postoje i biblioteke koje pojednostavljuju otkrivanje svojstava, od kojih je zasigurno najpopularnija Modernizr, sa jednostavnim APIjem za otkrivanje svojstava:

```
if (
    /* ako preglednik podržava <canvas> i canvas text? */
    Modernizr.canvas && Modernizr.canvastext &&
    /* preglednik podržava <video> i H.264 ofrmat? */
    Modernizr.video && Modernizr.video.h264 &&
    /* preglednik podržava local storage i application cache? */
    Modernizr.localstorage && Modernizr.applicationcache &&
    /* preglednik ... webworkers*/
    Modernizr.webworkers &&
    /* preglednik ... websockets*/
    Modernizr.websockets &&
    /* preglednik podržava Geolocation API? */
    Modernizr.geolocation &&
    /* preglednik podržava History API? */
    Modernizr.history
) {...}
```

HTML5 nastoji reducirati broj atributa u različitim elementima, pa tako <DOCTYPE> deklaracija više ne zahtjeva cijeli naziv i poveznicu na tekst standarda, a <html> zahtjeva samo deklararaciju jezika i kulture. Meta deklaracije seta znakova, linkovi zaglavlja (CSS, RSS, alternativni jezici i formati i slično), deklaracija relativnog značenja stranice i ostali elementi koji su postojali i u ranijim verzijama standarda sada su olakšani od svih nepotrebnih atributa.

Nova HTML specifikacija sadrži i nove semantičke elemente za grupiranje vrsta sadržaja. Iako je u trenutku pisanja prvenstvena praktična prednost olakšavanje stilizacije, njihova je bitnija namjena olakšano strojno razumijevanje sadržaja s ciljem bolje pretrage, indeksiranja i automatizirane prilagodbe sadržaja novim medijima. <section> element koristi se za tematsko grupiranje sadržaja, pri čemu <hgroup> sadrži naslov ili naslovni segment sekcije. <article> element predstavlja samostalnu jedinicu sadržaja, pri čemu se može sastojati od više sekcija. <aside> tipično predstavlja sadržaj koji je tematski relativno povezan sa svojom okolinom, ali odvojen grafičkom prezentacijom. <nav> elementi grupiraju navigaciju, zaglavlja označavamo elementom <header>, dok <footer> elementom označavamo podnožja. Važno je napomenuti da se zaglavlja i podnožja, ovisno o njihovoj relativnoj poziciji u DOM-u odnose na različite elemente, pa tako <header> čiji je roditelj <body> ima značenje zaglavlja dokumenta, dok <header> čiji je roditelj <article> ima značenje zaglavlja članka.

Naslovi dokumenta (<h1> do <h6>) također ima također imaju drugačiju semantiku: HTML4 dozvoljava samo jedan <h1> - odnosno glavni naslov stranice, dok manji naslovi (<h2> - <h6>) predstavljaju daljnju jednoznačnu hijerarhijsku strukturu dokumenta. Kod HTML5, <h1> predstavlja glavni naslov određenog roditelja, što znači da jedna stranica može imati neograničeno mnogo naslova <h1>, a hijerarhija naslova više nije postojana. Time se omogućuje dodavanje elemenata na stranicu koji sadrže vlastite naslove bez narušavanja hijerarhije naslova. Kako bi se omogućilo strojno čitljivo zapisivanje datuma i vremena, uveden je novi <time> element. Označavanje dijelova teksta sa ciljem dodjele značaja ili promjene stila, sada se vrši elementom <mark>.

```
<!DOCTYPE html>
<html lang="hr">
<head>
  <title>Primjer HTML5 markupa</title>
  <link rel="stylesheet" type="text/css" href="Content/style.css" />
  <link rel="alternate" type="application/atom+xml"
        title="Moj Atom feed" href="/feed/" />
  <link rel="shortcut icon" href="/favicon.ico">
  <script src="Scripts/jquery-1.9.1.js"></script>
</head>

<body>
```

```
  <header>
    <img src="logo.png" alt="Logotip"/>
    <hgroup>
      <h1>Naslov stranice</h1>
       <h2>Podnaslov</h2>
    </hgroup>
  </header>
  <nav>
    <ul>
      <li><a href="#">Početna</a></li>
      <li><a href="#">Blog</a></li>
      <li><a href="#">Članci</a></li>
      <li><a href="#">Galerija</a></li>
    </ul>
  </nav>
  <article>
    <hgroup>
      <h1>Naslov članka</h1>
      <h2>Podnaslov članka</h2>
    </hgroup>
    <section>
      <h1>Naslov prve sekcije</h1>
      <p>
        Tekst prve sekcije. <time datetime="2013-03-01">Prošlog
        petka</time> otvorila su se vrata percepcije. Ostati će
        otvorena <mark>do sljedeće obavijesti</mark>.
      </p>
    </section>
    <section>
      <h1>Naslov druge sekcije</h1>
      <p>Tekst druge sekcije.</p>
    </section>
  </article>
</body>
</html>
```

Canvas je kontejner element za programatsko iscrtavanje dvodimenzionalne grafike u web pregledniku. Canvas API omogućuje proceduralno iscrtavanje linija, oblika i tekstura na površinu, bez stvaranja grafa scene – odnosno logičke strukture crteža koja bi omogućila navigaciju, manipulaciju stvorenim objektima i spremanje strukture u memoriju za buduće korištenje. Za razliku od Canvasa, SVG (Scalable Vector Graphics) sadrži graf scene i pojedini elementi mogu reagirati na događaje. Međutim, nedostatak SVG-a je mnogo veća alokacija resursa, zbog čega se SVG koristi samo kada je takva funcionalnost uistinu potrebna.

<video> element omogućuje prikaz video sadržaja unutar web stranice ili aplikacije, odnosno reprodukciju unutar preglednika, međutim postoji polarizacija između proizvođača preglednika u odabiru podržanih formata video kontejnera. Microsoft i Apple zastupaju korištenje H.264/MPEG-4 AVC kontejnera, koji iako tehnički napredan i vrlo raširen, nije slobodan odnosno iziskuje plaćanje naknade za korištenje patentiranog formata. Proizvođači Mozilla i Opera podržavaju slobodan Theora/Ogg format, koji je na Appleov zahtjev izbačen iz HTML5 specifikacije kao zahtjevani format na temelju potencijalnih tehničkih problema u ostvarivanju kompatibilnosti na mobilnim uređajima, te zbog rizika od tužbe na temelju nepoznatih patenata. Google je problemu slobodnog video formata pristupio akvizicijom tvrtke On2 – vlasnice VP8 video formata, te promijenio uvjete korištenja kako bi format postao slobodan za korištenje bez naknada. Googleov WebM kontejner, koji može sadržavati VP8 video i Vorbis audio materijale tako je postao većinski podržan u preglednicima Chrome, Android browser, Chromium, Firefox i Opera, međutim Microsoft i Apple u trenutku pisanja rada još nisu usvojili navedene slobodne formate. Slična situacija odnosi se i na audio formate, pri čemu Apple i Mirosoft podržavaju isključivo patentirane MP3 i AAC formate, dok Google, Mozilla i Opera podržavaju Ogg Vorbis, WebM Vorbis i Ogg Opus slobodne formate.

HTML5 Geolocation API pruža sučelje za određivanje geografske lokacije i kretanja korisnika. Potaknuti sigurnosnim implikacijama i privatnošću korisnika, trenutna W3 HTML5 radna specifikacija nalaže da preglednik mora eksplicitno tražiti dopuštenje za korištenje geolokacijskih podataka od korisnika prilikom prvog korištenja, odnosno zahtjeva za korištenjem na određenoj domeni. Samu lokaciju moguće je redoslijedom rastuće prezicnosti odrediti koristeći IP adresu korisnika, korištenu bežičnu mrežu odnosno njezinu pristupnu točku, korišteni mobilni odašiljač ili GPS navigacijski modul ugrađen u mobilni uređaj.

W3 Geolocation API radni standard zahtjeva od preglednika minimalno četiri parametra: vrijeme uzimanja uzorka, zemljopisnu širinu, zemljopisnu dužinu i preciznost mjerenja, odnosno toleranciju na grešku izraženu u metrima. Neobavezni dopunski parametri su nadmorska visina, te brzina i smjer kretanja koji se mogu odrediti prema korisnikovom prethodnom položaju. API također dozvoljava ulazni parametar tražene preciznosti, pa će tako u slučaju zatražene niske preciznosti na mobilnim uređajima koristiti celularnu triangulaciju – nepreciznu, ali energetski

štedljivu tehnologiju, dok će u slučaju tražene visoke preciznosti upotrijebiti ugrađeni GPS navigacijski modul kako bi postigla traženu preciznost uz veći energetski trošak. U trenutku pisanja ovog rada, Apple iOS i Google Android operacijski sustavi omogućuju odvojeno postavljanje ovlasti za nisku i visoku preciznost geolokacije po pojedinoj domeni. Geolokacijski zahtjevi uvijek su asinhroni jer nije moguće jamčiti brzi odziv kada isti zahtjeva korisničku dozvolu i pobudu odgovarajućih uređaja ili mehanizama za geolokaciju.

```
$(function () {
    if (Modernizr.geolocation) {
        var options = {
            enableHighAccuracy: false, /* tražena razina preciznosti */
            timeout: 10000, /* maksimalno vrijeme odziva (ms) */
            maximumAge: 10000 /* maksimalna starost uzorka (ms) */
        };
        navigator.geolocation.getCurrentPosition(mojGeoCallback,
           mojErrorCallback)
    } else {
        alert('Vaš preglednik ne podržava HTML5 Geolocation API.')
    }
});

function mojGeoCallback(arg) {
    var vrijeme = arg.timestamp;          /* vrijeme uzorkovanja */
    var sirina = arg.coords.latitude;     /* zemljopisna širina */
    var duzina = arg.coords.longitude;    /* zemljopisna dužina */
    var preciznost = arg.coords.accuracy; /* preciznost uzorka */
    var visina = arg.coords.altitude;     /* nadmorska visina */
    var smjer = arg.coords.heading;       /* smjer kretanja */
    var brzina = arg.coords.speed;        /* brzina kretanja */
}

function mojErrorCallback(error) {
    var kod = error.code;
    /* Greška u prijenosu može biti:
     * 1 permission denied
     * 2 position unavailable – network down / satellites offline
     * 3 timeout
     */
    var poruka = error.message;
    alert('Greška ' + kod + ': ' + poruka)
}
```

5.2. CSS3 i CSS *frameworks*

Cascading Style Sheets (CSS) 3 kolokvijalni je naziv za skup različitih standarda i funkcionalnosti stvorenih nakon unificirane W3C CSS 2.1 specifikacije, podijeljenih u više od 50 modula koje održava W3C CSS radna grupa. Za potrebe ovog rada, istaknuti ćemo samo neke mogućnosti novih specifikacija, većinom implementirane u popularnim preglednicima:

- web fontovi i integracija ne-nativnih fontova
- novi sustavi za pozicioniranje (Grid Layout, Flexbox)
- prozirnost (en. *opacity)* i nove notacije boja
- prijelazi boja (en. *gradients*)
- sjenčanje teksta i pravokutnih elemenata (en. *shading*)
- napredna kontrola pozadinske grafike
- transformacije
- tranzicija i animacija

Web font specifikacia definira sintaksu za povezivanje fontova koji nisu instalirani na klijenskom računalu, odnosno koji su dostupni putem URL-a i privremeno instalirani za prikaz web stranice koja ih integrira. CSS3 specifikacija definira nekoliko različitih formata: EOT – Embedded OpenType, WOFF – Web Open Font Fort, SVG –Scalable Vector Graphic Font, TTF – TrueType Font, te OTF – OpenType Font formate, od kojih su većinski podržani WOFF, te TTF i OTF.

CSS3 pruža dva nova sustava za pozicioniranje elemenata: Grid Layout i Flexible Box Model - Flexbox, koji definiraju nove modele dinamičkog pozicioniranja vidljivih elemenata DOM strukture. Grid strukture relativno su slične <table> strukturama, međutim bez semantike koja implicira tabularnu strukturu sadržaja i sa dodanim mogućnostima poput slojeva i preklapanja. Za razliku od Grida koji je u trenutku pisanja podržan samo u Internet Explorer 10, Flexbox je djelomično ili potpuno podržan u svim popularnim preglednicima. Mogućnosti Flexboxa uključuju vertikalno centriranje, definiranje redoslijeda elemenata izvan markupa, i rastresite (en. *loose*) definicije koje omogućuju automatsku promjenu pozicija i veličina elemenata ovisno o veličini ekrana, omogućujući tako brži razvoj responzivnog web dizajna (en. *responsive web*).

Prozirnost (en. *opacity)* je osvojstvo koje je moguće dodijeliti većini vidljivih DOM elemenata u posljednjim verzijama svih popularnih preglednika, većinom u dostupnoj skali intenziteta 0/0.01 (potpuno prozirno) do 100/1 (potpuno neprozirno). Uz postojeće CSS 2.1 RGB i nazivne (ključne riječi) notacije boja, tako se uvodi RGBa notacija sa četvrtom dimenzijom prozirnosti u skali 0-1 preciznosti 0.01. Uz RGB, CSS3 specifikacija uvodi i HSL (Hue, Saturation, Lightness), odnosno HSLa notacije za zapis boja. Prednost HSL-a je intuitivnost, odnosno lakše ljudsko predviđanje promjena boja sa promjenom neke od dimenzija.

Sjenčanje teksta (en. *text shadows*) i pravokutnih elemenata (en. *box shadows*) dostupno je u svim modernim preglednicima, te u oba slučaja omogućuje horizontalno i vertikalno pozicioniranje sjene, intenzitet i veličinu, te boju sjene.

U napredne kontrole pozadinske grafike uključujemo eksplicitno definiranje veličine pozadine – koje može biti različito od veličine korištene slike, početnu točku pozadine (u odnosu na početak unutrašnjosti kontejnera, njegovih margina ili ispunjavanja), te mogućnost korištenja neograničenog broja različitih (i različito pozicioniranih) pozadina na istom elementu. Većina naprednih kontrola pozadinske grafike dostupna je u trenutku pisanja rada u svim vodećim web preglednicima.

Transformacijama možemo dvodimenzionalno ili prostorno rotirati (en. *rotate*), preseliti (en. *translate*), skalirati (povećati ili smanjiti) i prostorni zakrenuti (en. *skew*) vidljivi DOM element. Transformacije prirodom CSS jezika mogu biti svojstvo određenog stanja objekta (na primjer, nalaženja pokazivača iznad objekta, ili klik nad objektom), ili svojstvo dodijeljeno elementu DOM manipulacijom korištenjem JavaScripta. Dvodimenzionalne i trodimenzionalne transformacije dostupne su korištenjem standardiziranih ili *vendor-specific* CSS selektora u svim vodećim preglednicima, a usvajanjem standarda ukloniti će se specifični prefiksi.

CSS transformacijske metode koriste se za 2D i 3D modifikacije koordinatnog prostora transformabilnih objekata i njihovih ugnježđenih pod-elemenata.

Transformabilni objekti su objekti čiji je raspored i smještaj određen CSS modelom bloka ili inline-level elementa, ili čije je display svojstvo tabličnog tipa, te određeni elementi unutar SVG

nazivnog prostora. Transformacijske metode u trenutku pisanja ovog rada su u eksperimentalnoj fazi, odnosno specifikacija je podložna promjenama, a u osnovne transformacije možemo ubrojiti translaciju, rotaciju, skaliranje i iskrivljavanje (en. *skew*). U naprednije rotacije ubrajamo matricu, zadanu transformacijskom matricom i translacijskim vrijednostima. Dvodimenzionalne i trodimenzionalne transformacije implementirane su u gotovo svim vodećim web preglednicima koristeći standardizirane ili specifične selektore, sa trendom migracije funkcionalnosti prema standardiziranim selektorima.

CSS tranzicije pružaju kontrolu nad trajanjima i krivuljama intenziteta animacija prilikom prijelaza DOM elemenata iz postojećeg u novi oblik. Prilikom odgovaraćujeg događaja koji uvjetuje promjenu CSS svojstava (klik, *hover*, promjena klase putem JavaScripta i slično), web preglednik na temelju deklaracije tranzicija izračunava međuoblike objekata i provodi animaciju odnosno prijelaz između dvaju stanja DOM elementa i njegovih pod-elemenata.

CSS animacije pružaju radni okvir za definiranje jednostavnih animacija objekata koristeći ključne slike (en. *keyframes*) za definiranje eksplicitnih međustanja objekata unutar vremenskih okvira trajanja animacije. U prednosti CSS animacije nad skriptnim, odnosno JavaScript animacijama ubrajamo jednostavnost korištenja, bolje performanse uslijed interne pregledničke optimizacije koja koristi tehnike poput preskakanja slika (en. *frame skipping*) i izvođenja bez iscrtavanja ukoliko prozor nije vidljiv, odnosno ukoliko se korisnik nalazi u drugoj stranici ili kartici preglednika. U globalna svojstva određene animacije ubrajamo vremensko kašnjenje, smjer, trajanje, te funkcije vremena (en. *timing functions*). Pojedine ključne slike, od kojih su obavezne početna i završna definiraju pojedina CSS svojstva odnosno atribute elementa.

Raznovrsnost današnjih popularnih preglednika (Chrome, Firefox, Internet Explorer, Safari, Opera, Android Browser), spojenih uređaja (desktop računala, mobilni telefoni, laptopi, tableti, igraće konzole i brojni drugi) i rezolucija ekrana, te tromost u razvoju novih CSS standarda i pregledničke podrške doprinijeli su stvaranju brojnih CSS *frameworka* čije su značajke:

- podrška za napredna pozicioniranja i raspored elemenata
- podrška za fluidne promjene pozicija, rasporeda i vidljivosti s ciljem stvaranja responzivnog weba
- novi elementi sučelja: navigacijske liste i izbornici, horizontalne i vertikalne kartice, *breadcrumb* navigacija, navigacija po stranicama, thumbnails, obavijesti i dijalozi, grupirani gumbi, gumi s padajućim izbornicima, i redizajnirani postojeći elementi dostupni u istom obliku u svim preglednicima
- mogućnost izmjene grafičke teme uz zadržavanje istog *markupa*
- integrirane ikone i alatni simboli
- napredna tipografija i integracija web fontova
- animacija UI elemenata i podrška za naprednu klijentsku (JavaScript) interakciju

U najpopularnije CSS frameworke ubrajamo Twitter Bootstrap, Blueprint, 960grid, HTML5 Boilerplate, Foundation i Skeleton, a značajna je i pojava *authoring* frameworka poput Compassa.

5.3. CSS predprocesiranje

Rastuća kompleksnost stilskih značajki modernih web stranica i aplikacija istaknula je mane postojećih specifikacija CSS jezika, kao što su nemogućnost uspinjanja po DOM strukturi, nepouzdanost vertikalnog pozicioniranja, nepostojanje varijabli i kalkulacija, globalni namespace svih stilova i nerazdvojnost stila i selektora. Usprkos nedostatcima, CSS je jedini stilski jezik implementiran u svim vodećim preglednicima, što čini uvođenje novog stilskog jezika nepraktičnim rješenjem. Djelomično rješenje nudi se u vidu stvaranja nadskupa CSS značajki čiji je izvorni kod moguće kompilirati u standardizirani W3 CSS 2.1, odnosno određeni dio W3 ili WHATWG CSS3 radne specifikacije.

LESS je unazad kompatibilno proširenje CSS jezika koje nudi značajke poput varijabli, funkcija, ugnježđenih pravila, uvjeta i drugih *helpera* koji omogućuju programerima pisanje stilske logike, odnosno kombinacije CSS stilova i programske logike koja se kompilira u standardizirani CSS, što je moguće automatizirati kao dio *build* skripte, ili prepustiti kompiliranje klijentima i

isporučivati .less datoteke do klijentske strane gdje će njihov preglednik preuzeti proces kompiliranja za vrijeme učitavanja stranice. Drugi pristup zahtjeva učitavanje LESS biblioteke za pred-procesiranje prije početka obrade, i preporučuje se samo u toku razvoja kada nije potrebno ponavljati *build* proceduru zbog sitnijih vizualnih izmjena.

LESS podržava varijable koje možemo koristiti za pohranu boja, dimenzija i sličnih podataka. Važna odlika LESS jezika jest *scope*, odnosno pripadnost varijable određenom bloku izvan kojeg je nedostupan (odnosno nedeklariran), stoga je potrebno obratiti pažnju na mjesto definicije.

```
@tamna: #1b1b1b;
@svijetla: #8e8e8e;
@rad: 5px;
.primjer {
    color: @tamna;
    background-color: @svijetla;
    border-radius: @rad;
}
```

Mixin, odnosno skup korisnički definiranih pravila obuhvaćen zajedničkim nazivom također je važan LESS koncept, budući da omogućava ponovno iskorištavanje cijelih stilskih blokova za različite selektore. Mixin može prihvaćati i parametre na temelju kojih će se vršiti transformacija stilskih značajki. Mixin može ili ne mora biti eksplicitno vidiljiv u kompiliranom CSS-u, odnosno ovisno o postojanju praznih zagrada u deklaraciji.

```
// Vidljivi mixin:
.mojStil {
    border: dashed solid #2bff00;
}
// Parametrizirani mixin:
.zaobljen (@radius: 5px) {
      -webkit-border-radius: @radius;
      -moz-border-radius: @radius;
      -ms-border-radius: @radius;
      -o-border-radius: @radius;
      border-radius: @radius;
}
// Nevidljivi mixin:
.nevidljiv {
    font: italic 22px Arial,Verdana;
}
```

```
// Upotreba mixina u selektoru:
#menu a{
    .mojStil;
    .nevidljiv;
    .zaobljen(10);
}
```

Ugnježđena pravila (en. *nested rules*) omogućuju primjenu stila na ugnježđenim elementima, odnosno praćenjem DOM strukture dokumenta. Ugnježđeni stilovi se također mogu iskoristiti na drugim mjestima koristeći znak '>' (veće od) za označavanje roditelj-dijete odnosa elemenata.

```
.citat {
      .zaobljen(10px);
      background: #bbb;
      font: normal 12px Georgia,serif;
      width: 370px;
      padding: 10px 10px 20px 10px;
      .autor {
            float: right;
            margin: 13px 5px 0 0;
            background: none;
      }
}
```

LESS podržava korištenje matematičkih operacija i usporedbi u stvaranju stilova, a kombiniranjem navedene funkcionalnosti mogu se stvoriti posebni uvjeti izvođenja. LESS sadrži i posebne funkcije za upravljanje bojama, dobivanje informacija o bojama i izvođenja operacija poput zaokruživanja brojeva. LESS također sadrži pomoćne metode za obavljanje transformacija boja i funkcije dohvaćanja informacija o bojama, poput svjetline, zasićenosti, količine boje određenog kanala i slično.

```
// Matematičke operacije
.primjer {
      color: @lightGrey / 2;
      font: bold 12px „Helvetica Neue“,arial,sans-serif;
      width: 400px - 10% + 2em;
}
// Posebne funkcije
```

```
@var: #bbb;
@daLiJeBoja: iscolor(@var);
@daLiJeUrl: isurl(@var);
@novaBoja: ligten(#ff6a00, 50%);
@drugaNovaBoja: mix(#4cff00, #b200ff, 10%);
@zasicenost: saturation(#ff006e);
@zaokruzeno: round(1.67);
// Uvjetovano izvođenje (guards)
@a: #000;   /*promjenom ove boje mijenja se pozadinska boja*/
.windowDesigner (@a) when (lightness(@a) >= 50%) {
      color: #000;
}
.windowDesigner (@a) when (lightness(@a) < 50%) {
      color: #fff;
}
.windowDesigner (@a) {
      background: @a;
      font: normal 10px Consolas,Courier,monospace;
      width: 390px;
      border: 1px dashed #000;
      margin: 10px 0 10px 0;
}
.window{
      .windowDesigner(@a);
}
```

6. Praktični rad: društvena mreža

S ciljem prikaza dostupnih alata i tehnologija razvoja modernih web aplikacija, ovaj rad prikazuje izradu pojednostavljene društvene mreže u koju se korisnici mogu prijaviti koristeći identitete vanjskih pružatelja usluga (Google, Facebook), održavati korisničke profile i dijeliti sadržaj sa različitim krugovima korisnika. Aplikacija će biti implementirana koristeći ASP.NET MVC4 *web framework* i SQL Azure distribuiranu bazu podataka, pri čemu će aplikacija i baza biti dostupne i replicirane putem Microsoft Azure cloud usluga. REST servisi namjenjeni asinkronom dohvatu sadržaja biti će napisani koristeći WebAPI biblioteku, dok će se putem SignalR *real-time web* biblioteke omogućiti dostava novog sadržaja korisnicima u trenutku njegovog stvaranja bez potrebe ponovnog učitavanja stranice ili vremenskim odmakom koji uobičajeno asociramo uz AJAX i srodne asinhrone tehnologije. Tokom razvoja također će se koristiti HTML5 za pisanje sloja pogleda i geolokaciju korisnika, te LESS za CSS stiliziranje izgleda aplikacije.

6. 1. Inženjerstvo zahtjeva

Inženjerstvo zahtjeva (en. *requirements engineering*) je proces izrade specifikacije programskog rješenja.[6] Prema razini detalja, razlikujemo korisničke zahtjeve, zahtjeve sustava i specifikaciju programske potpore. Korisnički zahtjevi i zahtjevi sustava razlikuju se prema sadržaju:

- funkcionalni zahtjevi – usluge sustava i njegova interakcija s okolinom
- nefunkcionalni zahtjevi
- zahtjevi domene primjene

Korisnički zahtjevi su specifikacija visoke razine apstrakcije napisana prirodnim jezikom i grafičkim dijagramima. Zahtjevi sustava su detaljna specifikacija sustava napisana strukturiranim prirodnim jezikom, posebnim jezicima za oblikovanje sustava, dijagramima i matematičkom notacijom. Specifikacija programske potpore najdetaljniji je opis koji objedinjuje korisničke

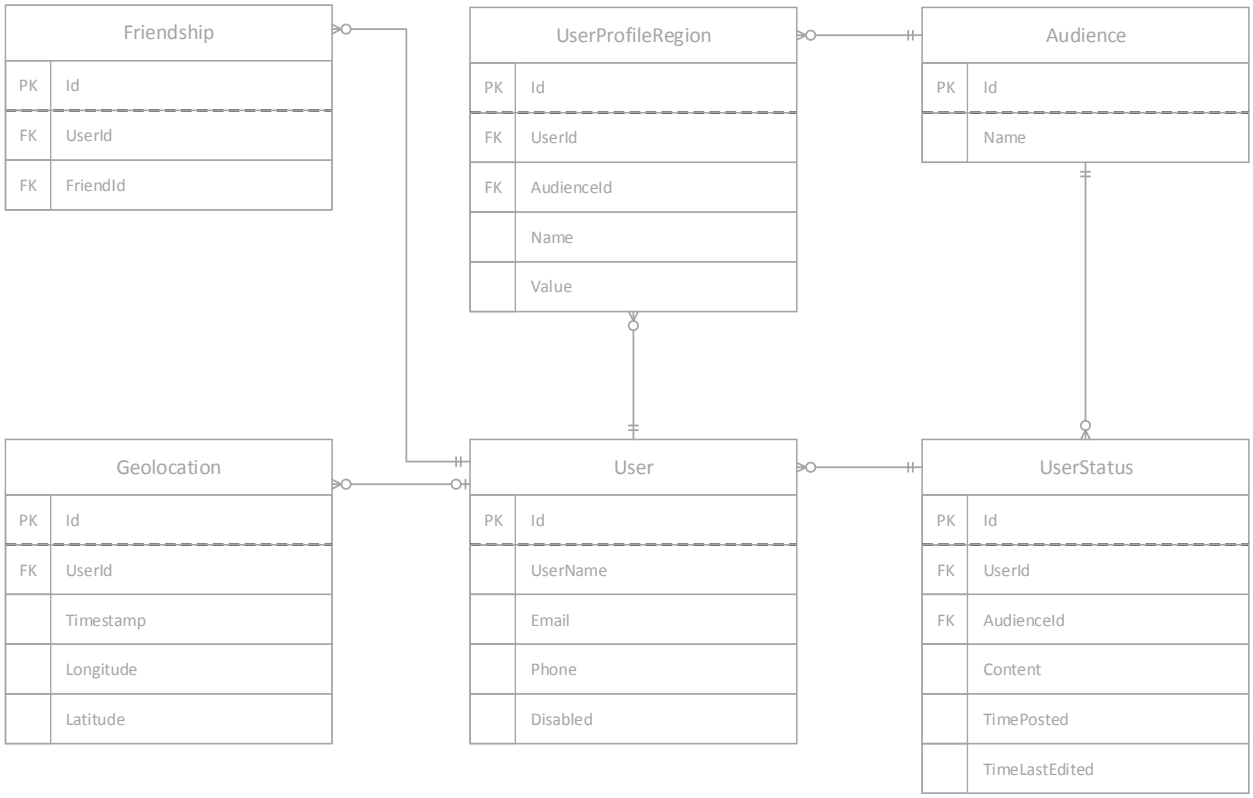
[6] Sommerville, I., Software engineering, 8th ed, Addison Wesley, 2007.

zahtjeve i zahtjeve sustava. U sklopu ovog rada, razmatrati ćemo funkcionalne korisničke zahtjeve:

- javna web aplikacija dostupna iz web preglednika bez nativnih komponenti
- registracija korisnika i mogućnost prijave sa postojećim Google ili Facebook korisničkim računima
- korisnici održavaju korisničke profile i statuse, koji mogu biti javni ili vidljivi samo prijateljima
- korisnički profili mogu se sastojati od različitih segmenata, koji posjeduju različite razine privatnosti
- prijateljstvo sa drugim korisnicima se postiže dodavanjem korisnika u vlastiti krug prijatelja – čime navedenom korisniku postaje vidljiv sadržaj namjenjen prijateljima
- korisnički profili imaju jasne javne adrese (na primjer: domena.hr/korisnik/ime.prezime)
- novi korisnički statusi vidljivi su u stvarnom vremenu prijateljima i/ili javnosti
- pretraga korisnika vrši se prema korisničkom imenu
- prilikom korištenja aplikacije, bilježi se geolokacija svih korisnika za daljnju analizu u drugim sustavima

6.2. Modeliranje podataka i procesa

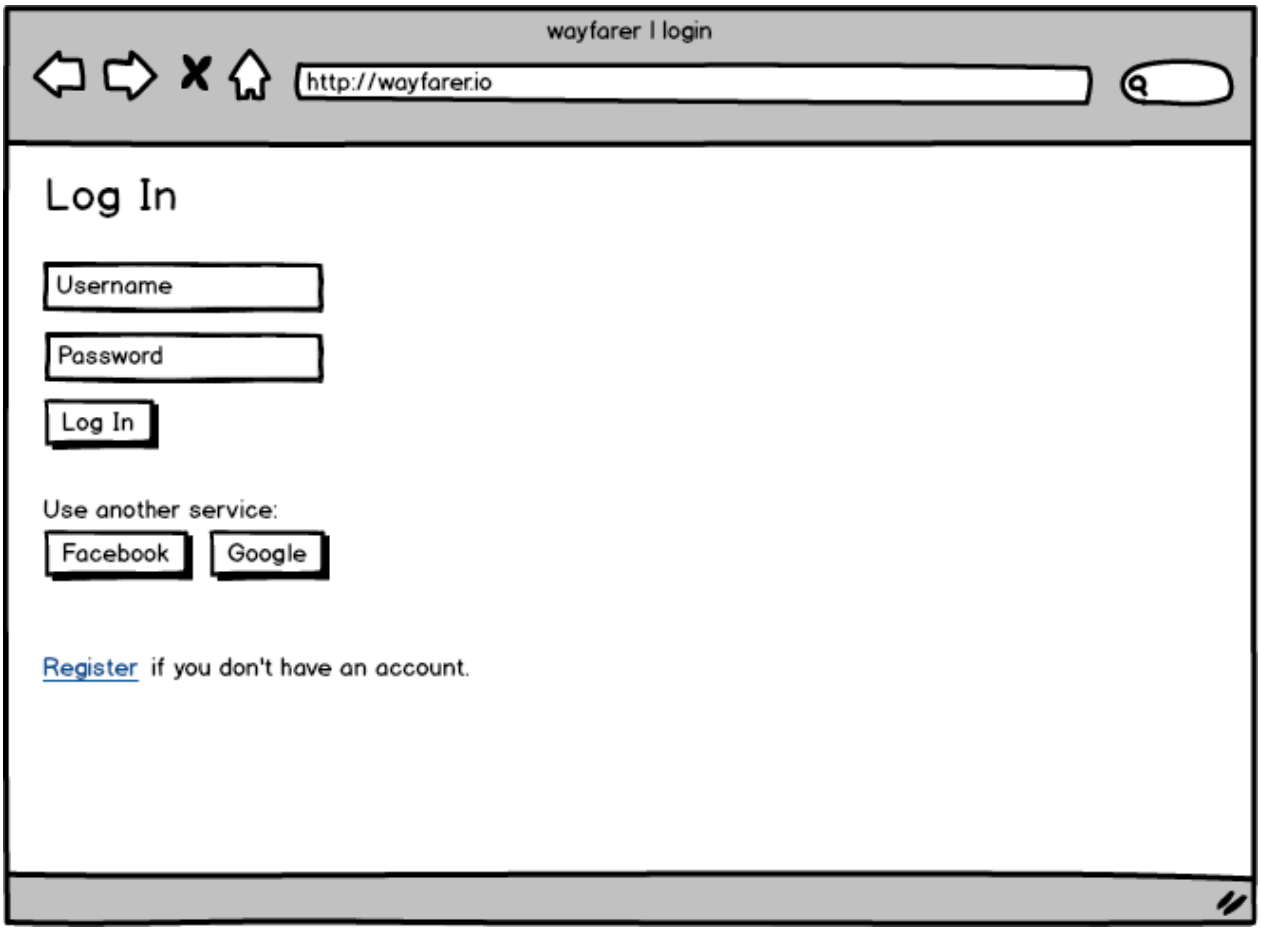
Na temelju funkcionalnih korisničkih zahtjeva moguće je odrediti temeljne entitete aplikacije, kao što su korisnik (User), korisnički status (User Status), regija korisničkog profila (User Profile Region), prijateljstva (Friendship), geolokacija (Geolocation), te namjenjena publika sadržaja (Audience). U nastavku prikazujemo dijagram Entiteti-veze navedenog sustava:



Slika 1: Dijagram Entiteti-veze društvene mreže

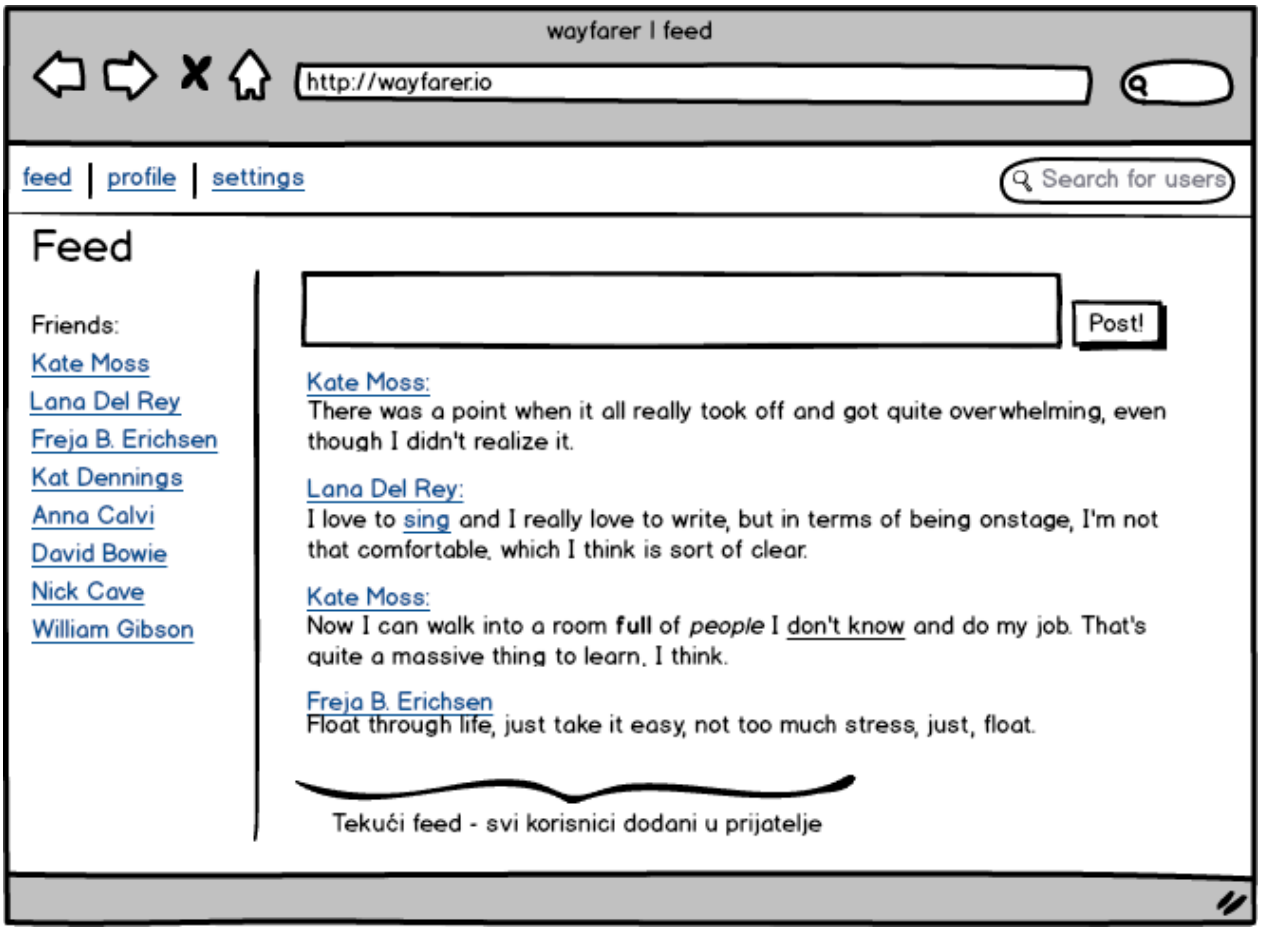
Sljedeći korak oblikovanja sustava jest izrada *wireframea*, odnosno prikaza kostura web aplikacije koji uključuje sadržaj, navigacijske elemente i općenito interakcijske elemente sučelja. *Wireframe* ne sadrži tipografiju, boje niti grafiku, budući da je njegova namjena prikaz funkcionalnosti i prioriteta pojedinog sadržaja.

Prvi ekran prikazuje prijavu korisnika putem korisničkog imena i lozinke specifične za web aplikaciju koju stvaramo, ili mogućnost prijave putem postojećih Facebook ili Google korisničkih računa. Ekran prijave sadrži i prečac na stranicu za registraciju novih korisnika.



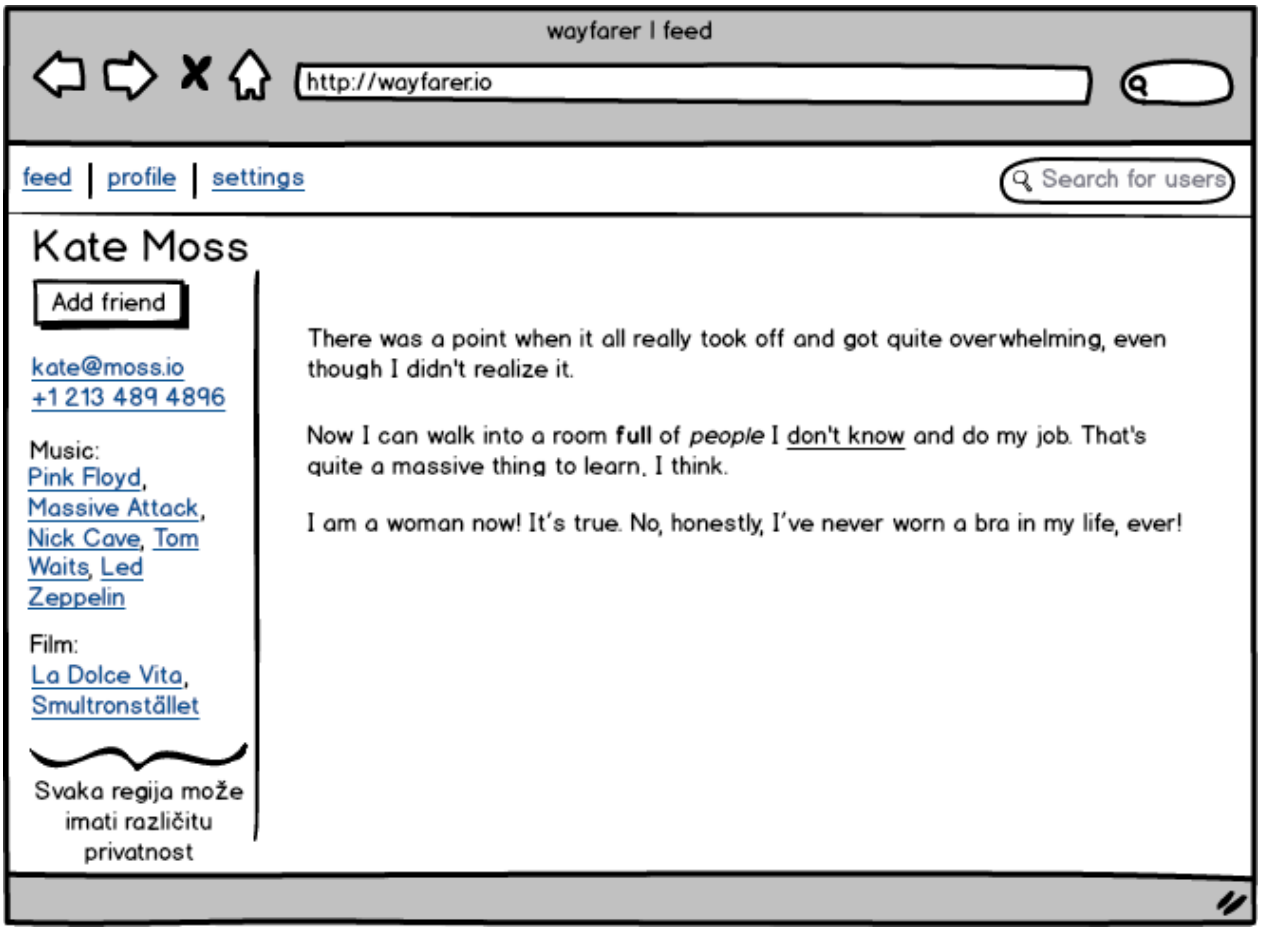
Slika 2: Dijagram za prijavu korisnika u društvenu mrežu

Sljedeći ekran prikazuje početnu stranicu nakon prijave. Na vrhu se nalazi glavni izbornik, koji omogućuje navigaciju na početnu stranicu, stranicu vlastitog profila, stranicu postavki te pretraživanje korisnika. Lijeva kolumna prikazuje listu korisnikovih prijatelja putem koje je moguće posjetiti pojedine korisničke profile. Na desnoj strani nalazi se forma za unos novog korisničkog statusa, te potom novi korisnički statusi korisnika dodanih na listu prijatelja. Ukoliko se korisnik nalazi na vrsti liste, prilikom pojave novog sadržaja (odnosno, kada netko sa liste prijatelja napiše novi status), novi će status prilikom nastanka biti dodan na početnu stranicu svih njegovih *online* prijatelja. Listanje statusa vrši se 'beskonačno', odnosno pomicanjem stranice prema dolje upiti se nastavljaju sve dok postoje kronološki stariji statusi prijatelja, odnosno dok postoji sadržaj koji je moguće učitati.



Slika 3: Početna stranice nakon prijave na društvenu mrežu

Posljednji ekran jest ekran profila: lijeva kolumna prikazuje detalje profila poput e-mail adrese, telefona i pojedinih regija koje mogu biti javne ili dostupne samo prijateljima. Web aplikacija ovisno o zahtjevatelju stranice određuje koje su regije i statusi dostupni. Desna kolumna prikazuje sve statuse navedenog korisnika, prikazane kronološki od najnovijeg prema najstarijem. Ukoliko se posjetitelj nalazi na vrhu stranice, novi sadržaj biti će prikazan u trenutku njegovog stvaranja, dok listanjem statusa prema dolje oni se nastavljaju ‘u beskonačnost’, odnosno do prvog statusa korisnika.



Slika 4: Ekran korisničkog profila društvene mreže

6.3. Izrada aplikacije

Nakon definiranja funkcionalnih korisničkih zahtjeva i modeliranja podataka i procesa, slijedi izrada aplikacije, započevši sa novim ASP.NET MVC4 projektom, instalacijom NuGet paketa i podešavanjem TFS sustava za upravljanje revizijama. Sljedeći je korak izrada podatkovnog sloja, odnosno definicija entiteta koristeći Entity Framework code first pristup i generiranje baze podataka i repozitorija za upravljanje podacima.

Korištenjem Dependency Injection tehnike, repozitoriji se vezuju za kontrolere namijenjenje isporučivanju tradicionalnih sinhronih sadržaja (glavne stranice, navigacije, statičkog sadržaja), REST servisa kojima će se asinhrono dohvaćati korisnički statusi i sličan sadržaj, te SignalR WebSockets kontrolera namijenjenih implementaciji real-time web funkcionalnosti. Klijentska

strana bazira se na HTML5 markupu, LESS-u te HTML5 API poput Geolication. Posljednji je korak deployment na Azure cloud uslugu koja omogućuje replikaciju i skaliranje web aplikacije i pripadajuće odnosno pripadajućih baza podataka.

6.3.1. Kontrola verzija i upravljanje paketima

Obavezni preduvjet izrade bilo koje moderne aplikacije jest uvođenje sustava kontrole verzija (en. *version control system*). Sustavi kontrole verzija mogu se podijeliti prema kriteriju modela repozitorija podataka na centralizirane (klijent-poslužitelj) i distribuirane (decentralizirane) sustave. Sustavi sa centralnim repozitorijem pohranjuju cjelokupnu povijest projekta na jednoj lokaciji, te na pojedine korisničke zahtjeve (en. *checkout*) dostavljaju samo tražene verzije datoteka. Jedna od prednosti takvog pristupa jest velika brzina dostave pojedinih zahtjeva, budući da se dostavlja samo najmanji skup izmjena. U nedostatke takvog pristupa možemo ubrojati povećani rizik od gubitka podataka (jedinstvena točka ispada) i obaveznu komunikaciju sa centralnim repozitorijem prilikom pretraživanja povijesti. U popularne centralizirane sustave kontrole verzija ubrajamo Subversion (SVN) i Team Foundation Server.

Distribuirani sustavi, kao što su Git i Mercurial osiguravaju preuzimanje cjelokupnog repozitorija sa prvim korisničkim zahtjevom, te preuzimanja novih segmenata repozitorija sa svakim sljedećim zahtjevom. Takav pristup čini sustav mnogo otpornijim na gubitke podataka, međutim može postati problematičan ukoliko se radi o velikim repozitorijima (reda veličine GB) zbog brzina prijenosa i memorijskih kapaciteta na pojedinim klijentima.

Za ovaj projekt odabran je Team Foundation Server, odnosno Microsoft Team Foundation Service usluga koja uključuje Team Foundation Server. Team Foundation Server izabran je iz tehnoloških i praktičnih razloga, odnosno nativne integracije sa Visual Studio razvojnom okolinom i mogućnostima praćenja zadataka (en. *tasks*) i prijave/rješavanja bugova.

Microsoft Team Foundation Service pruža cjelokupno rješenje za vođenje projekta i pridruženih repozitorija putem web sučelja, te korištenje istih u Visual Studio razvojnoj okolini. Usluga pruža i potpunu podršku za upravljanje projektom:

- praćenje napretka prema Scrum, Agile ili CMMI metodologiji vođenja projekta i planiranje kapaciteta
- unošenje i praćenje zadaća i prijavljenih bugova sa detaljnim opisima, procjenama trajanja i dodjelama pojedinim developerima ili fazama projekta – sprintevima u slučaju Agile metodologije
- automatizirani build projekta i izvođenje unit testova nakon check-ina
- automatizacija deployment procedura na Windows Azure
- pretplata na slanje notifikacija za određene događaje (check-in, greška u kompiliranju itd.)
- podrška za proces recenzije koda sa mogućnostima komentiranja pojedinih linija
- integracija sa Windows Live identitetima
- integracija sa Microsoft PowerPoint za storyboarding funkcionalnost

Visual Studio 2012 omogućuje upravljanje paketima koristeći NuGet. Korištenjem Package Manager konzole moguće je instalirati potrebne projekte u Visual Studio ASP.NET MVC projekt:

```
PM> Install-Package jQuery
Successfully installed ‘jQuery 1.9.1’.
Successfully added ‘jQuery 1.9.1’ to Wayfarer.Samples.

PM> Install-Package Modernizr
Successfully installed ‘Modernizr 2.6.2’.
Successfully added ‘Modernizr 2.6.2’ to Wayfarer.Samples.
```

Odabirom Solution – Add Solution to Source Control u Visual Studio razvojnoj okolini dodajemo naše rješenje sa svim pripadajućim datotekama i konfiguracijom paketa na Team Foundation Server. Sljedeći paket koji dodajemo je Dotless, LESS predprocesor za .NET uključen u NuGet Package Manager. Podrazumijevani način rada dotless-a jest kompiliranje LESS izvorne datoteke na poslužiteljskoj strani na klijentski zahtjev, te isporuka rezultirajuće CSS datoteke. Autor ovog rada preferira kompiliranje jednom, odnosno dodavanje naredbi za kompilaciju u *Pre-build events* projekta:

```
del “$(ProjectDir)Content\style.css”
“$(SolutionDir)packages\dotless.1.3.1.0\tool\dotless.compiler.exe” –m
```

```
"$(ProjectDir)Content\style.less" "$(ProjectDir)Content\style.css"
```

Nova verzija Microsoft Visual Studio razvojne okoline u pravilu stiže svake tri do četiri godine, što prestavlja vrlo značajan vremenski period u svijetu web aplikacija. Iz tog razloga, Microsoft objavljuje takozvani ASP.NET and Web Tools set dodataka kojima nastoje pratiti sve novitete na području web aplikacija. Posljednja verzija, odnosno verzija 2012.2 uključuje novu verziju tehnologija ASP.NET Web Forms i ASP.NET MVC, te omogućuje pisanje RESTful servisa baziranih na ASP.NET Web API tehnologiji. Također Web Tools 2012.2 uključuje SignalR – novu Microsoftovu biblioteku za real-time web komunikaciju, te naglašavanje sintakse (en. *syntax highlighting*) za TypeScript i LESS datoteke.

6.3.2. Podatkovni sloj: entiteti, baza podataka i repozitoriji

Upotrebom Entity Framework *code-first*, moguće je definirati entitete i njihove veze kao C# klase, te potom generirati bazu podataka koristeći Code First Migrations. Slijedi prikaz dvaju izrađenih i povezanih entiteta te konteksta putem kojeg se omogućuje korištenje:

```
[Table("UserProfile")]
public class UserProfile
{
    /*opisni atributi (anotacije) primarnog ključa*/
    [Key]
    [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]
    public int UserId { get; set; }
    public string UserName { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public bool Disabled { get; set; }
    /*veza prema UserProfileRegion entitetima*/
    public virtual ICollection<UserProfileRegion> Regions { get; set; }
}
public class UserProfileRegion
{
    [Key]
    [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]
    public int UserProfileRegionId { get; set; }
    public UserProfile User { get; set; }
    public Audience Audience { get; set; }
    public string Name { get; set; }
    public string Value { get; set; }
```

```
}
/*kontekst svih entiteta sustava*/
public class WayfarerContext : DbContext
{
    public WayfarerContext() : base("DefaultConnection") {...}
    public DbSet<UserProfile> UserProfiles { get; set; }
}
```

Dependency Injection tehnika omogućuje zamjenu komponenti repozitorija tokom izvršavanja (en. *at runtime*). Takva se funkcionalnost postiže pisanjem sučelja (en. *interface*) koje sadrži sve javne metode repozitorija, te zatim instanciranja odgovarajućeg repozitorija (koji implementira navedeno sučelje) u trenutku instanciranja kontrolera:

```
/*sučelje repozitorija*/
public interface IGeolocationRepository
{
    void StoreGeolocation(string ip, decimal longitude, decimal latitude,
        string username = null);
}

/*implementacija sučelja*/
public class GeolocationRepository : IGeolocationRepository
{
    WayfarerContext _context;
    public GeolocationRepository()
    {
        _context = new WayfarerContext();
    }
    public void StoreGeolocation(string ip, decimal longitude,
        decimal latitude, string username = null)
    {
        _context.Geolocations.Add(new Geolocation(){...});
        _context.SaveChanges();
    }
}

/*injekcija repozitorija u kontroleru*/
public class HomeController : Controller
{
    private IProfileRepository _repository;
    private Microsoft.AspNet.SignalR.IHubContext _wayfarerHub;
    public HomeController(IProfileRepository repository)
    {
        this._repository = repository;
        this._wayfarerHub =
```

```
            Microsoft.AspNet.SignalR
                .GlobalHost.ConnectionManager
                .GetHubContext<WayfarerHub>();
        }
    public HomeController() : this (new ProfileRepository() ){...}
    public ActionResult Index() {...}
}
```

Jedna od prednosti Entity Framework ORM sustava nad izravnom komunikacijom sa bazom podataka jest mogućost korištenja LINQ upitnog jezika nad entitetima, zaobilazeći tako SQL u potpunosti. Pisanje kompleksnih upita postaje mnogostruko jednostavije:

```
public List<Status> GetFeed(string requestor, int? skip = null, int?
beforeId = null, int? afterId = null)
{
    var publicIds = _context.Friendships
        .Where(f => f.Profile.UserName == requestor)
        .Select(s => s.Friend.UserId).ToList();
    var friendshipIds = _context.Friendships
        .Where(f => f.Friend.UserName == requestor)
        .Select(s => s.Profile.UserId).ToList();
    friendshipIds.Add(_context.UserProfiles
        .First(u => u.UserName == requestor).UserId);
    IEnumerable<Status> query = null;

    if (beforeId == null && afterId == null)
        query = _context.Statuses.Include("Author").Where(s =>
            (publicIds.Contains(s.Author.UserId)
            && s.Audience == Audience.Public) ||
            (friendshipIds.Contains(s.Author.UserId)
            && s.Audience == Audience.Friends)
    );
    else if (beforeId != null) {...}
    else if (afterId != null) {...}

    if (skip == null)
        return query.OrderByDescending(o => o.Id)
                    .Take(Config.QueryLimit).ToList();
    else
        return query.OrderByDescending(o => o.Id)
                    .Skip(skip.Value).Take(Config.QueryLimit).ToList();
}
```

6.3.3. Kontroleri, WebAPI servisi i SignalR

ASP.NET MVC tradicionalni kontroleri instanciraju se prilikom HTTP(S) zahtjeva. Uobičajeno, oni instanciraju repozitorij koji provodi manipulaciju podataka i dostavlja rezultate obrade u repozitorij. Dohvaćenim podacima manipulira se na način da se stvara ViewModel ili prenose u ViewBag, te se potom tokom parsiranja pogleda integriraju u pogled koji biva dostavljen korisniku. Uz tradicionalne kontrolere, aplikacija koristi i WebAPI, te SignalR kontrolere. WebAPI je ASP.NET *framework* namijenjen pisanju REST HTTP servisa, koji će se koristiti za dohvat korisničkih statusa prilikom pregleda početne stranice (*global feed*) ili pojedinog korisničkog profila. SignalR je biblioteka namijenjena dodavanju *real-time web* funkcionalnosti ASP.NET aplikacijama, te će u ovom radu biti korištena za notifikacije o stvaranju novog sadržaja kako bi pravovremeno bio dohvaćen koristeći skripte klijentske strane i REST servise poslužiteljske strane. U nastavku slijedi prikaz korištenih tradicionalnih, WebAPI i SignalR kontrolera:

```
/*tradicionalni ASP.NET MVC kontroler*/
public class ProfilesController : Controller
{
    private IProfileRepository _repository;
    public ProfilesController(IProfileRepository repository)
    {
        this._repository = repository;
    }
    public ProfilesController() : this (new ProfileRepository() ){...}

    [Authorize]
    public ActionResult Index(string id)
    {
        var profile = _repository.GetProfileByUsername(id);
        var regions = _repository.GetRegions(id, User.Identity.Name);
        ViewBag.Profile = profile;
        ViewBag.Regions = regions;
        return View();
    }
}
```

```
/*WebAPI kontroler*/
public class FeedController : ApiController
{
    private IProfileRepository _repository;
    public FeedController(IProfileRepository repository)
    {
        this._repository = repository;
    }
    public FeedController() : this (new ProfileRepository() ){...}

    public IEnumerable<FeedItem> Get(string username, int? skip = null,
                                     int? beforeId = null,
                                     int? afterId = null)
    {
        var results = new List<FeedItem>();
        if (User.Identity.IsAuthenticated)
        {
            var feed = _repository.GetUserStatuses(username,
                           User.Identity.Name,
                           skip, beforeId, afterId);
            results = GetFeedViewModel(feed, _repository);
        }
        return results;
    }
}

/*SignalR kontroler*/
public class WayfarerHub : Hub
{
    private IProfileRepository _profileRepo;
    private IGeolocationRepository _geolocRepo;
    private ISearchRepository _searchRepo;

    public WayfarerHub(IProfileRepository profileRepo,
                       IGeolocationRepository geolocRepo,
                       ISearchRepository searchRepo)
    {
        this._profileRepo = profileRepo;
        this._geolocRepo = geolocRepo;
        this._searchRepo = searchRepo;
    }

    public WayfarerHub() : this (new ProfileRepository(),
                                 new GeolocationRepository(),
                                 new SearchRepository() ){...}

    public override Task OnConnected()
    {
```

```
        SubscribeToFriendEvents();
        return base.OnDisconnected();
    }

    public bool SaveGeolocation(decimal longitude, decimal latitude)
    {
        var ip = HttpContext.Current.Request.UserHostAddress;
        var user = Context.User.Identity.IsAuthenticated ?
                   Context.User.Identity.Name : null;
        return _geolocRepo.StoreGeolocation(ip, longitude, latitude, user);
    }
}
```

6.3.4. Klijentska strana: HTML5, LESS i geolokacija

Kako bi ubrzali razvoj aplikacije, kao osnovni HTML5 *framework* odabran je **Twitter Bootstrap**, trenutno u nestabilnoj verziji 3.0 RC1. Twitter Bootstrap nudi brojne komponente i funkcionalnosti koje ubrzavaju razvoj *front-end* dijela, kao što su *grid* sustav, ikone, tipografija, tablice, forme i elementi formi, navigacija, gumbi, straničenje, te JavaScript događaji i tranzicije. Bootstrap također omogućuje brže pisanje responsivnih stranica inherentnom podrškom za mobilne uređaje. Framework dolazi u kompajliranoj (CSS) ili nekompajliranoj (LESS) varijanti, pri čemu je potrebno koristiti LESS kompajler prilikom pokretanja *build* procedure. Korištena verzija LESS kompajlera naziva se **dotless**, dostupna putem Visual Studio NuGet *package managera*

Twitter Bootstrap ne posjeduje vlastiti *templating engine*, pa je u tu svrhu odabran **Handlebars.js**, kako bi se podaci (konkretno, korisnički statusi) dohvaćeni putem REST servisa grafički oblikovali na klijentskoj strani prije samog prikaza. Sama DOM manipulacija izvodi se uz pomoć biblioteke **jQuery**, dok otkrivanje mogućnosti preglednika vrši biblioteka **Modernizr**. U nastavku slijedi prikaz relevantnih Twitter Bootstrap komponenti i JavaScript koda korištenog na glavnoj stranici društvene mreže:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>Home | Wayfarer | Wayfarer</title>
```

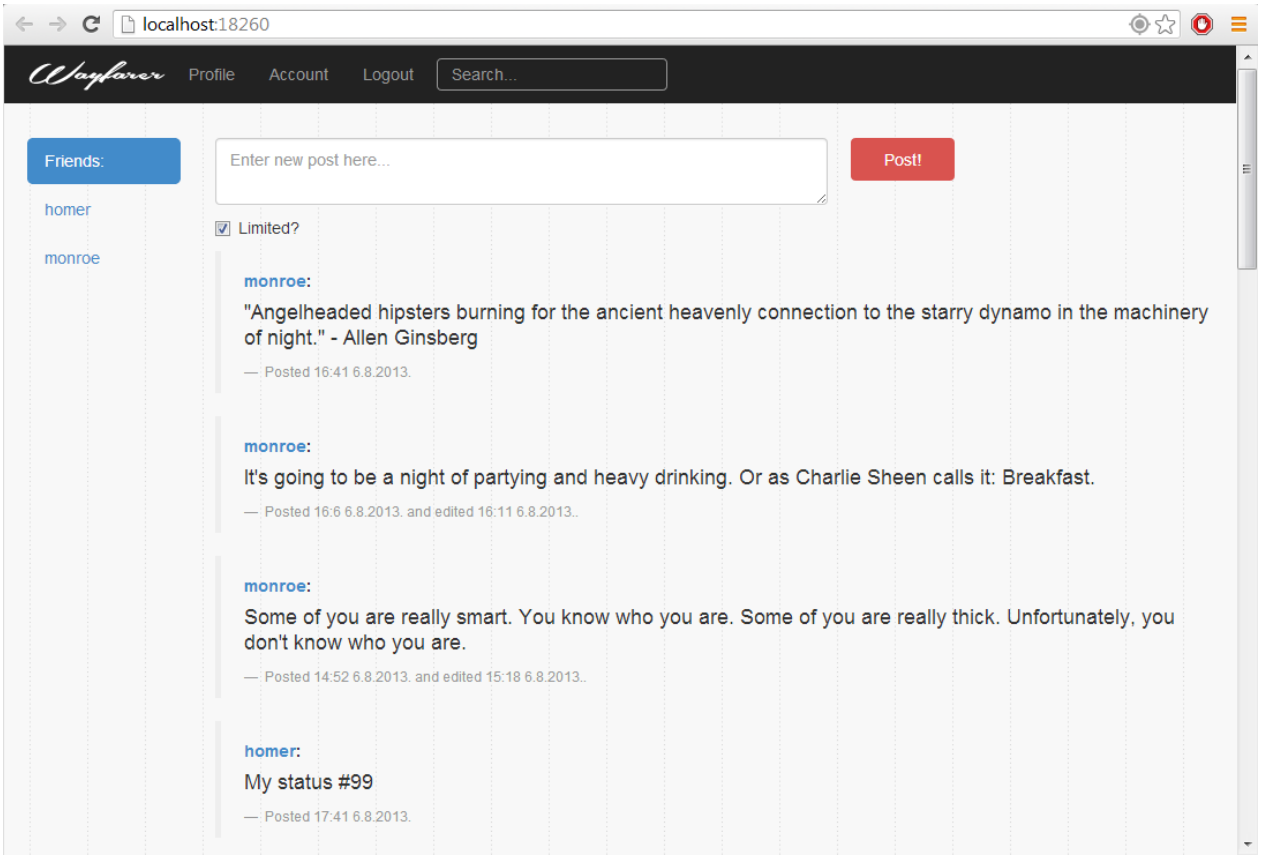
```
<!-- skaliranje na različitim uređajima: -->
<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<!-- CSS datoteke - u produkciji spojene i minimizirane -->
    <link href="/Content/bootstrap/bootstrap-glyphicons.css"
rel="stylesheet"/>
    <link href="/Content/bootstrap/bootstrap.css" rel="stylesheet"/>
    <link href="/Content/site.css" rel="stylesheet"/>
</head>

<body>
<!-- navigacija: -->
<div class="navbar navbar-fixed-top navbar-inverse">
    <a class="navbar-brand" href="/">
        <img src="/Content/images/wayfarer_logo.png" alt="Wayfarer"/></a>
    <ul class="nav navbar-nav">
    <li><a href="http://localhost:18260/profiles/tinc2k">Profile</a></li>
    <li><a href="/Account/Manage">Account</a></li>
    <li><a href="/Account/Logout">Logout</a></li>
    <li class="dropdown">
    <!-- pretraga i kontejner rezultata -->
    <input type="search" autocomplete="off" name="search"
           class="form-control" placeholder="Search..." />
    <ul class="dropdown-menu dropdown-search" role="menu"
        aria-labelledby="dropdownMenu"></ul>
    </ul>
</div>

<!-- sadržaj:-->
<div class="row">
    <!-- Twitter Bootstrap grid - lijeva kolumna -->
    <div class="col-6 col-lg-2">
        <!-- lista prijatelja: -->
        <ul class="nav nav-pills nav-stacked">
            <li>...</li>
        </ul>
    </div>
    <!-- Twitter Bootstrap grid - desna kolumna -->
    <div class="col-12 col-lg-10">
        <div id="status_post">
            <!-- forma za unos novog statusa -->
            <form class="form">
                <textarea>...<textarea>
                <button type="submit"
                        class="btn btn-danger btn-lg">Post!</button>
            </form>
        </div>
        <!-- kontejner statusa -->
```

```
        <div id="status_container"></div>
    </div>
</div>


<!-- Handlebars.js predložak (template) pojedinog statusa -->
<script id="status-template" type="text/x-handlebars-template">
    <blockquote class="status">
        <strong><a
href="{{status.AuthorUrl}}">{{status.AuthorName}}</a>:</strong>
        <p>{{status.Content}}</p>
        <small>
            Posted {{status.TimePosted}}
            {{#if status.TimeEdited}}
            and edited {{status.TimeEdited}}.
            {{else}}
            {{/if}}
        </small>
    </blockquote>
</script>

<!-- JavaScript biblioteke - u produkciji spojene i minimizirane -->
<script src="/Scripts/jquery-2.0.0.js"></script>
<script src="/Scripts/modernizr-2.6.2.js"></script>
<script src="/Scripts/jquery.signalR-1.0.1.js"></script>
<script src="/Scripts/bootstrap.js"></script>
<script src="/Scripts/handlebars.js"></script>
<script src="/signalr/hubs"></script>

<script type="text/javascript">

/*globalne varijable*/
var search_container;
var status_container
var search_template;
var status_template;
var wayfarerHub = null;
var statuses = new Array();

/*događaj: učitavanje dokumenta (pojednostavljeno)*/
$(function () {
    search_container = $('.dropdown-search');
    status_container = $('#status_container');
    /*kompiliranje Handlebars.js predloška*/
    search_template = Handlebars.compile($("#search-template").html());
    status_template = Handlebars.compile($("#status-template").html());
    /*početno učitavanje statusa*/
    getFeed(null, null, null, forceWriteTop);
```

```
    /*inicijalizacija SignalR WebSockets veze s poslužiteljem*/
    wayfarerHub = $.connection.wayfarerHub;
    $.connection.hub.start().done(function () {
        geolocateMe(); /*započni slanje geolokacije pri uspostavi*/
        window.setInterval(geolocateMe, 30000);
    });
    /*definicija SignalR metoda*/
    wayfarerHub.client.refreshFeed = function () {
        var latest_status = getLatestStatusId();
        getFeed(null, null, latest_status, OnScrollOrUpdate)
    }
});

/*događaj: scroll (pojednostavljeno)*/
$(window).scroll(function () {
    OnScrollOrUpdate();
});

/*pretraživanje (pojednostavljeno)*/
$("input[name='search']").keyup(function (e) {
    $.ajax(
    {
        url: "/api/search",
        contentType: "text/json",
        data: { query: value },
        success: function (data) {
            /*punjenje memorije dohvaćenim rezultatima*/
            $.each(data, function (index) {
                searchData.push(...);
            });
            for (var i = 0; i < searchData.length; i++) {
            /*kompilacija putem predloška i dodavanje u DOM*/
            $(search_container).append(
                search_template({ searchItem: searchData[i]})
            );
        }
    });
});

/*zapis novih statusa na vrh stranice (pojednostavljeno)*/
function forceWriteTop() {
    var latest_id = getLatestStatusId();
    var new_statuses = new Array();
    for (var i = 0; i < statuses.length; i++)
        if (latest_id == undefined || statuses[i].StatusId > latest_id)
            new_statuses.push(statuses[i]);
    for (var i = 0; i < new_statuses.length; i++)
        $(status_container).prepend(
```

```
            status_template({status: new_statuses[i]})
        );
}

/*zapis starijih statusa na dno stranice (pojednostavljeno)*/
function forceWriteBottom() {
    var oldest_id = getOldestStatusId();
    for (var i = 0; i < statuses.length; i++)
        if (statuses[i].StatusId < oldest_id)
            $(status_container).append(
                status_template({status: statuses[i]})
            );
}

/*pri pomicanju stranice ili dolasku novog sadržaja*/
function OnScrollOrUpdate() {
    if ($(window).scrollTop() < $(status_container).offset().top)
        forceWriteTop();
    else if ($(window).scrollTop() + $(window).height() >=
$(status_container).offset().top + $(status_container).height()) {
        var oldest_id = getOldestStatusId();
        getFeed(null, oldest_id, null, forceWriteBottom);
    }
}
/*dohvaćanje statusa (pojednostavljeno)*/
function getFeed(skip, beforeId, afterId, callback) {
    $.ajax(
    {
        url: "/api/feed",
        contentType: "text/json",
        data: { skip: skip, beforeId: beforeId, afterId: afterId },
        success: function (data) {
            $.each(data, function (index) {statuses.push(...);});
            callback();
        }
    });
}
/*geolokacija (pojednostavljeno)*/
function geolocateMe() {
    if (Modernizr.geolocation) {
        var options = {
            enableHighAccuracy: true, timeout: 10000, maximumAge: 10000
        };
    navigator.geolocation.getCurrentPosition(
            geolocateMeCallbackSuccess, geolocateMeCallbackError);
    }
}
function geolocateMeCallbackSuccess(arg) {
```

```
    wayfarerHub.server.saveGeolocation(
            arg.coords.longitude, arg.coords.latitude);
};
</script>
</body>
</html>
```



Slika 5: Početna stranica društvene mreže

6.3.5. Deployment aplikacije na Windows Azure

Windows Azure skup *cloud* usluga je zasigurno najpopularniji izbor *hostinga* za .NET web aplikacije. Planiranje poslovnog kontinuiteta, odnosno stabilnosti i replikacije web aplikacija i baza podataka na Windows Azure kompleksan je posao koji uključuje brojne izbore, te ćemo za potrebe ovog rada samo napomenuti kako Azure omogućuje upravljanje prometom odnosno preusmjeravanje zahtjeva na određenu instancu kriterijima geografske lokacije, opterećenja i

stanja instance, te mnogo varijanti ponovne uspostave usluge od reinstalacije virtualnog stroja i aplikacije do aktivno-pasivnih i aktivno-aktivnih konfiguracija pojedinih instanci aplikacije.

Sve Azure SQL baze podataka podešene za *high availability* sadrže jednu primarnu i dvije sekundarne replike baze. Sva čitanja i pisanja u bazu podataka izvršavaju se na primarnoj replici te bivaju propagirane na sekundarne replike, međutim sve transakcije zahtjevaju potvrdu većine, odnosno potvrdu primarne i barem jedne sekundarne replike prije odluke o uspješnosti transakcije. U slučaju pada primarne replike, vanjski arbitrar određuje jednu od sekundarnih replika kao novu primarnu repliku kriterijem manjeg opterećenja poslužitelja na kojem se replika nalazi. U slučaju pada sekundarne replike, kriterij uspješnosti transakcije raste, odnosno zahtjeva potvrdu primarne i opstale sekundarne replike za potvrdu transakcije. Vanjski arbitrar određuje stupanj kvara replike te odlučuje da li je repliku moguće popraviti (softverski *update*, restart i slično) ili je potrebno stvoriti novu repliku i kopirati sadržaj baze podataka sa primarne replike. Velikom povezanošću i međusobnim kontrolama, svaki Azure datacentar može izgubiti do 15% svih poslužitelja i zadržati razinu usluge iznad one određene korisničkim ugovorima.

Deployment aplikacije i baze podataka na Windows Azure iz Visual Studio 2012 razvojne okoline iznimno je jednostavan zahvaljujući dostupnosti *deployment* konfiguracijske datoteke za odgovarajuće Web Site i Database instance prilikom korištenja Azure web sjedišta. Slijedi primjer navedene konfiguracijske datoteke:

```
<publishData>
<publishProfile profileName="Tin Crnković - Wayfarer Deployment"
    publishMethod="MSDeploy"
    publishUrl="wayfarer.publish.azurewebsites.windows.net:443"
    msdeploySite="tin.crnkovic" userName="$tin.crnkovic"
    userPWD="password" destinationAppUrl="http://wayfarer.io"
    SQLServerDBConnectionString="
        Data Source=tcp:wayfarer.database.windows.net,1433;
        Initial Catalog=wayfarerDb;
        User ID=tin.crnkovick@wayfarer;
        Password=password"
    controlPanelLink="http://windows.azure.com"
    targetDatabaseEngineType="sqlazuredatabase"
    targetServerVersion="Version100">
    <databases>
    <add name="wayfarerDb"
```

```
        connectionString="
            Data Source=tcp:wayfarer.database.windows.net,1433;
            Initial Catalog=wayfarerDb;
            User ID=tin.crnkovic@wayfarer;
            Password=password"
        providerName="System.Data.SqlClient"
        targetDatabaseEngineType="sqlazuredatabase"
        targetServerVersion="Version100"/>
    </databases>
</publishProfile
```

7. Zaključak

Rast kvantitete i tehnološke raznolikosti raširenih umreženih uređaja, te rastuće potrebe ciljanih skupina korisnika uvelike mijenjaju funkcionalne i tehničke kriterije koje moderna web aplikacija mora zadovoljiti. Iz navedenih razloga, nove tehnologije i alati ne rješavaju samo ranije poznate probleme već uvode i nove funkcionalnosti poput geolokacije i interakcije sa vanjskim uslugama.

Iz perpektive podatkovnog sloja, najznačajnija je promjena odmicanje od tradicionalnih relacijskih baza podataka, odnosno raslojavanje dijela podataka na nestrukturirane i slabo strukturirane odnosno ključ-vrijednost parirane podatke. Takva promjena potaknuta poboljšavanjem performansi i smanjenjem troškova cloud usluga uvodi novi izazov u smislu održavanja podataka web aplikacije konzistentnima ili djelomično konzistentnima, pa zasigurno možemo ustvrditi kako kompleksnost stvaranja i održavanja podatkovnog sloja iz perspektive modeliranja i izrade raste.

Mogućnosti modernih web *frameworka* kao što su objektno-relacijsko preslikavanje, agnostični upitni jezici, upravljanje rutama, parsiranje predložaka i brojne druge stvaraju isključivo prednost u smislu lakoće i brzine izrade web aplikacije. Također je važno istaknuti jednostavnost pisanja REST servisa, koji uvelike zamjenjuju njihove SOAP ekvivalente u svim scenarijima izuzev implementacije etabliranih sigurnosnih protokola.

Razvoj klijentskih JavaScript biblioteka u iznimnom je porastu, pa tako danas JavaScript uzima prvo mjesto na zajednici GitHub sa 21% udjela među ostalim programskim jezicima. Istaknuti ćemo samo neke od najpopularnijih klijentskih biblioteka, kao što su jQuery, Backbone, Foundation, AngularJS, Modernizr i ember.js. Pojava poslužiteljskih JavaScript *frameworka* i biblioteka otvara po prvi put mogućnost razvoja web aplikacije u jednom programskom jeziku na poslužiteljskoj i klijentskoj strani, što smanjuje barijeru ulaska novim programerima i omogućuje korištenje jednog formata pohrane (JSON) od NoSQL baze podataka poput MongoDB do bogatih klijentskih biblioteka poput AngularJS. Uz izniman broj biblioteka, vrlo važna pojava jesu alati za upravljanje JavaScript paketima poput Bowera, te alate za automatizaciju radnog procesa poput Grunta. Zahvaljujući tromosti u razvoju nove ECMAScript specifikacije, javljaju se jezici

koje je moguće transkompilirati u JavaScript poput CoffeeScript-a i TypeScript-a, koji olakšavaju pisanje jezika programerima koji imaju iskustva u drugim jezicima (Ruby, Python, C#), što je također prednost u odnosu na ranije dostupne alate i tehnologije.

HTML5 i povezani standardi u razvoju predstavljaju veliki potencijal, međutim izuzev nekoliko vrlo uspješnih tehnologija poput Geolocation API i Web Storage, mogućnosti su još uvijek uvelike ograničene trenutno dostupnim web preglednicima – posebno u pogledu video i audio formata, te izravne komunikacije između klijenata (WebRTC). Situacija s HTML5 dodatno je otežana nepostojanjem zajedničkog plana razvoja i različitom razinom podržanosti na popularnim uređajima, pa tako dodatnu kompleksnost s klijentske strane uvodimo otkrivanjem mogućnosti i stvaranjem alternativnog sadržaja u slučaju nepodržanosti na web pregledniku. Slične argumente moguće je donijeti i za CSS3 i vezane standarde za koje također ne postoji zajednički plan usvajanja između proizvođača web preglednika, koji pak uvođenjem vlastitih prefiksa za određene funkcionalnosti zasigurno ne olakšavaju proces razvoja i testiranja aplikacija u različitim preglednicima.

Biblioteke za *real-time web* komunikaciju poput SignalR i socket.io olakšavaju izradu web aplikacija koje komuniciraju u stvarnom vremenu pružajući API koji je tehnološki agnostičan, te se brinu za odabir najpovoljije komunikacijske tehnologije koju podržavaju klijent i poslužitelj. Možemo reći kako će navedene i slične tehnologije vjerojatno u potpunosti zamijeniti AJAX u nadolazećim godinama budući da pružaju veće tehničke mogućnosti uz istu lakoću implementacije.

Biblioteke za akceleraciju *front-end* razvoja, kao što su Twitter Bootstrap, HTML5 Boilerplate i Foundation značajno ubrzavaju razvoj web aplikacija u slučajevima u kojima dizajn nije prethodno strogo definiran. U tom pogledu, zbog rastućih je kriterija (responzivnost, performanse) *front-end* razvoj vrlo specifičnih zahtjeva mnogo teži, dok su generički zahtjevi mnogo jednostavniji zahvaljujući postojećim *front-end* bibliotekama.

U konačnici, naglasiti ćemo kako su mogućnosti modernih tehnologija i alata razvoja web aplikacija iznimno napredovale u odnosu na one dostupne prije nekoliko godina, međutim sa

njihovim rastom značajno se mijenjaju znanja i vještine koje programer mora posjedovati kako bi riješio izazove koje ekosustav predstavlja.

8. Popis literature

1. Freeman et. al., Head First Design Patterns, O'Reilly Media, digitalno izdanje, 2004.
2. Esposito, D., Saltarello, A., Architecting Microsoft .NET Solutions for the Enterprise, Microsoft Press, digitalno izdanje, 2008.
3. Armbrust et al., A View of Cloud Computing, Communications of the ACM, Vol. 53 No. 4, 2010. (http://cacm.acm.org/magazines/2010/4/81493-a-view-of-cloud-computing/fulltext)
4. Sadalage P. J., Fowler, M., NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Addison-Wesley, digitalno izdanje, 2012.
5. Becker, R., Windows Azure Programming Patterns for Start-ups, Packt Publishing, digitalno izdanje, 2012.
6. Klein, S., Roggero, H., Pro SQL Database for Windows Azure, Apress, digitalno izdanje, 2012.
7. Griffiths, I., Programming C# 5.0, O'Reilly Media, digitalno izdanje, 2013.
8. Troelsen, A., Pro C# 5.0 and .NET 4.5 Framework, Apress, digitalno izdanje, 2012.
9. Albahari, J., Albahari, B., C# 5.0 in a Nutshell: The Definitive Guide, O'Reilly Media, digitalno izdanje, 2012.
10. Arsenovski, D., Professional Refactoring in C# and ASP.NET, Wrox, digitalno izdanje, 2009.
11. Galloway et. al., Professional ASP.NET MVC 4, Wiley Publishing, digitalno izdanje, 2012.
12. Flanagan, D., JavaScript The Definitive Guide, O'Reilly Media, digitalno izdanje, 2011.
13. Freeman, A., Pro JavaScript for Web Apps, Apress, digitalno izdanje, 2012.
14. Freeman, A., Pro jQuery, Apress, digitalno izdanje, 2012.
15. McFarland, D. S., JavaScript and jQuery - The Missing Manual, O'Reilly Media, digitalno izdanje, 2012.
16. Osmani, A., Learning JavaScript Design Patterns, O'Reilly Media, digitalno izdanje, 2012.
17. Joshi, B., HTML5 Programming for ASP.NET Developers, Apress, digitalno izdanje, 2012.
18. Freeman, A., The Definitive Guide to HTML5, Apress, digitalno izdanje, 2011.
19. Pilgrim, M., HTML5 Up and Running, O'Reilly Media, 2010.
20. Frain, B., Responsive Web Design with HTML5 and CSS3, Packt Publishing, digitalno izdanje, 2012.
21. Atwood, J., Effective Programming: More Than Writing Code, Coding Horror, digitalno izdanje, 2012.
22. Swizec, T., Why Programmers Work At Night, Lean Publishing, digitalno izdanje, 2012.