UNIVERZITET U BEOGRADU

MATEMATIČKI FAKULTET

NOVI PRISTUP KONSTRUKCIJI VEB APLIKACIJA

MASTER RAD

Mentor:

prof. dr Vladimir Filipović

Kandidat:

Aljoša Šljuka

Sadržaj

1. Uvod

Predmet istraživanja ovog rada su moderne Veb aplikacije, prije svega one koje se izvršavaju u Veb pregledaču. Glavni motiv su tzv. „Veb aplikacije u jednoj strani“, čiji naziv je prevod engleskog termina *Single Page Applications*[1]. Zbog glomaznosti termina i prevoda, kroz rad će se “aplikacije u jednoj strani” referisati pomoću već uvriježenog akronima originalnog termina, SPA. Cilj ovog rada je predstaviti SPA pristup pri konstrukciji Veb aplikacija, njegove prednosti nad tradicionalnim Veb aplikacijama i izazove koji proističu korištenjem ovog pristupa. Pored toga, u radu će biti predstavljena generalna anatomija jedne SPA, inspirisana najboljim praksama i potkrijepljena konkretnim primjerima.

Tradicionalne Veb aplikacije ne distribuiraju procesiranje informacija i klijenta koriste isključivo kao korisnički interfejs. U njima svaki zahtjev korisnika pokreće ponovno generisanje stranice, koja se čitava svaki put ponovo dostavlja korisniku. Ovaj proces sadrži mnogo ponavljanja i opterećuje serverski sistem. SPA je Veb aplikacija koja svu navigaciju vrši u jednoj strani, većinu poslovne logike distribuira na klijenta, a server koristi za trajno čuvanje podataka, implementaciju sigurnosti i dostavljanje podataka i statičnih datoteka klijentu.

U radu će se kao primjer koristiti aplikacija za rukovanje rasporedima nastave. Ona je prvobitno razvijena kao projekat za predmet Razvoj softvera 2 na master studijama modula Računarstvo i informatika smjera Matematika Matematičkog fakulteta Univerziteta u Beogradu, a u svrhu pomoći rada više službi Filozofskog fakulteta Univerziteta u Istočnom Sarajevu. Kroz rad će biti prikazani najvažniji djelovi njenog dizajna, uključujući izvorni kôd, često modifikovan zbog jasnoće ili bolje ilustracije koncepta. Ovoj Veb aplikaciji se može pristupiti na adresi: http://sekretarski-modul.apphb.com/.

[1] Još poznate i kao *Single Page Interfaces* ili SPI.

2. Generacije Veba i razvoj Veb aplikacija

"Ne postoji činjenica koja se činjenicom može nazvati,
a da nije na ploči za podatke negdje u nekom rezervoaru -
osim ako je u pitanju činjenica koju tehničari upravo iskopavaju i stavljaju na ploču"
- Logika imena Džo
Marej Lenister

World Wide Web (poznat i kao *WWW, Web* ili *W3*) je skup međusobno povezanih hipertekstualnih dokumenata, kojima se pristupa putem Interneta. Internet je globalni sistem međusobno povezanih računarskih mreža koje koriste standard Internet protokola (*TCP/IP*) da povežu nekoliko milijardi uređaja širom svijeta. To je *mreža svih mrežâ*, koja se sastoji od miliona privatnih, javnih, akademskih, poslovnih i državnih mreža, lokalnog i globalnog opsega, koje su povezane nizom elektronskih, bežičnih i optičkih mrežnih tehnologija.

Veb aplikacija je klijent/server aplikacija koja koristi Veb pregledač kao klijentski program i izvršava interaktivne usluge povezujući se sa serverima preko Interneta [1].

Da bi se pravilno prikazale osobine modernih Veb aplikacija, potrebno ih je postaviti u istorijski kontekst razvoja samog Veba i brojnih tehnologija koje ga čine. Stoga, rad počinje kratkim pregledom razvoja Veba.

Jedna od prvih ideja za ono što danas zovemo Vebom datira još od 1946. i pripovjetke "Logika zvana Džo" Mareja Lenistera[2]. Tu su opisani uređaji, zvani "logikama", koji se nalaze u svakoj kući i povezani su na distribuirani sistem repozitorijuma (nazvanih "rezervoarima"). Na njima se čuva svaki podatak koji je ljudski rod zabilježio, te se po potrebi, i sa dozvolom, djeli ljudima preko „logika". Lenister prikazuje ideju ogromne informacione mreže dostupne svima, te šta bi sve moglo poći po zlu ako se zloupotrijebi. Naime, jedna logika je zbog tehničke greške počela sama da pretražuje rezervoare, dostupne informacije obrađuje i iz njih vadi nova saznanja, do tada nepoznata čovjeku, te naravno, informacije dalje dijeli ljudima. *"Da ugasimo rezervoar? Da li ti je možda palo na pamet, druže, da rezervoar obavlja sva izračunavanja za sve kompanije već godinama? Obavlja distribuciju devedeset četiri posto svih TV programa, daje sve*

[2] *Murray Leinster* (1896. - 1975.) - američki pisac naučne fantastike.

informacije o vremenu, rasporedima letova, posebnim sniženjima, mogućim zapošljenjima i vjestima; upravlja svom ličnom komunikacijom preko žice i bilježi sve poslovne razgovore i ugovore - Slušaj, druže! Logike su promjenile civilizaciju. Logike jesu civilizacija!" [2]. Zastrašujuće predviđanje...

No, pravo ostvarenje Veba je došlo tek nekih 45 godina kasnije, a 20 godina nakon što je prva veza uspostavljena putem onoga što je danas poznato kao Internet. Tim Berners Li[3] je tada radio kao softverski inžinjer u *CERN*-u[4], gdje su mnogi naučnici provodili istraživanja, a potom ih nastavljali u sopstvenim laboratorijama širom svijeta. Prirodno, željeli su da djele podatke i rezultate istraživanja, s čim su imali velike poteškoće. Berners je uvidio taj problem, kao i ogromni neiskorišteni potencijal Interneta. U to vrijeme je radio na informacionom sistemu koji je upravljao hipertekstualnim dokumentima - dokumentima koji su, pored teksta, sadržavali i veze prema drugim dokumentima, te tako omogućavali laku navigaciju s jednog na drugi. Berners je napravio server za distribuiranje ovih dokumenata, a kasnije, udruživši snage sa Roberom Kaijuom[5], prvi Veb pregledač, nazvan *WorldWideWeb*, te *URL*[6] i *HTTP*[7] tehnologije. 6. avgusta 1991. lansirana je prva Veb stranica[8], a popularnost Veba se širila kroz naučne zajednice. No, događaj koji je zaista napravio eksploziju popularnosti Veba, i vjerovatno uzrokovao da postane ono što je danas, desio se 30. aprila 1993. Tada je *CERN* objavio izvorni kôd tehnologija u osnovi Veba, uključujući i *WorldWideWeb* pregledač.

U tom periodu začetka Veba, Veb stranice su bile striktno tekstualni dokumenti, sa dodatkom hipertekstualnih veza. *HTML* je sadržavao opis za veliki broj tagova koji se koriste i danas: H1-6 (zaglavlja), P (paragraf), A ("sidro", veza prema drugom dokumentu), UL (lista bez nabrajanja), OL (lista sa nabrajanjem), LI (element liste), BLOCKQUOTE (blok za citiranje) itd. Stranice su imale zadatak da prenesu informacije i reference prema drugim i taj zadatak su obavljale efikasno.

---

[3] *Tim Berners Lee* (1955. -) britanski informatičar.

[4] Evropski savjet za nuklearno istraživanje (fr. *Conseil européenne pour la recherche nucléaire*) .

[5] *Robert Cailliau* (1947. -) belgijski informatičar.

[6] *Uniform Resource Locator* - sprecifičan niz karaktera koji predstavlja jedinstvenu referencu na resurs.

[7] *Hypertext Transfer Protocol* - aplikativni protokol za distribuirane, kolaborativne, informacione sisteme zasnovane na hipermedijima.

[8] Kopija prve Veb stranice se može naći na http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html.

Do kraja 1993. *NSCA*[9] na *Unix*, *Macintosh* i *Windows* sisteme donosi Veb pregledač *Mosaic*, a sa njim i sasvim novo iskustvo pregedanja Veba. Donio je podršku za prikazivanje slika unutar Veb strane sa IMG tagom, te podršku za zvuk i video, a elementi korisničkog interfejsa koje je nudio (linija za *URL*, dugmad za navigaciju naprijed, nazad i osvježavanje) su i dan danas obavezni dijelovi svakog modernog Veb pregledača. Prema riječima Gerija Vulfa (*Gary Wolf*), novinara *Wired* magazina[10]: "Šarmantni izgled *Mosaic*-a ohrabruje korisnike da postave svoje sopstvene dokumente na Veb, uključujući slike u boji, zvučne i video snimke… *Mosaic* nije najdirektniji način dolaska do *online* informacija. Niti je najmoćniji. On je prosto *najprijatniji*, i u 18 mjeseci otkako je izdat, *Mosaic* je potakao navalu uzbuđenja i komercijalne energije kojoj nema premca u istoriji Veba" [3]. I zaista, za tu godinu i po postojanja *Mosaic*-a, broj Veb stranica je sa 130 skočio na 10.022 [4]!

Veb više nije bio ekskluzivan za naučnu zajednicu, a Veb stranice su sa čisto tekstualnih prelazile na zabavne, šarene sadržaje, zanimljive širim krugovima konzumenata. Konstrukcija Veb stranica je zahtjevala sve više dizajniranja, te se mogao vidjeti začetak onoga što je danas unosni posao Veb dizajnera [5].

Broj tehnologija vezanih za Veb je vrtoglavo rastao. Oktobra 1994. Berners osniva *World Wide Web Consortium* (*W3C*), međunarodnu organizaciju za standardizaciju Veba. I to u pravo vrijeme, jer se do kraja 1995. vodio pravi rat pregledača. Autori originalnog *Mosaic*-a napuštaju *NCSA* i osnivaju *Netscape Communications Corporation*, koja je decembra 1994. izdala prvu verziju *Netscape Navigator*-a. *Spyglass Inc.* (komercijalna grana *NCSA*) je licencirala *Mosaic Microsoft*-u, što je predstavljalo osnovu za *Internet Explorer*, čija je prva verzija izdata avgusta 1995. *Telenor*, norveški telekomunikacioni gigant, je finansirao projekat *Opera*, iz kog se aprila 1995. izrodila prva verzija istoimenog pregledača.

Pregledači su se pokazali kao interesantna platforma, te je brzo nastao čitav niz tehnologija koje proširuju njihove osnovne mogućnosti. Najznačajnije među njima su *JavaScript* i *CSS* (*Cascading Style Sheets*) tehnologije. *JavaScript* (prvobitno izdat kao *LiveScript*) razvijen je od strane *NetScape*-a i pružio je mogućnost upravljanja pregledačem i dinamičkog modifikovanja *HTML* dokumenta na klijentu programskim putem. Prihvaćen je (sa izmjenama) od strane *Microsoft*-a kao *JScript*, a kasnije standardizovan kao *ECMAScript*. *CSS* je nastao kao

[9] *National Center for Supercomputing Applications* - američko partnerstvo za razvoj nacionalne sajberinfrastrukture.
[10] *Wired* - američki mjesečni štampani i *online* magazin koji uglavnom izvještava o uticaju novih tehnologija na kulturu, ekonomiju i politiku.

spoj dva jezika za opis stilova *HTML* dokumenata od devet koji su predloženi u *W3C www-style* mejling listi. Pružao je konzistentan način opisivanja stilskih osobina *HTML* dokumenta[11], te omogućio jasno razdvajanje strukture od izgleda dokumenta.

Na žalost programera, proizvođači pregledača su se više trudili da uvedu nove mogućnosti, uključujući specifične *HTML* tagove, *CSS* selektore i semantičke osobine skriptnih jezika, nego da poprave već postojeće. Obezbjediti iste karakteristike Veb stranice na različitim pregledačima je često bilo veoma teško, pa su se čak razvijale zasebne Veb stranice za prikaz u različitim pregledačima.

Sa druge strane, izrodio se čitav niz serverskih tehnologija, koje su obrađivale zahtjeve klijenata i isporučivale dinamički generisane dokumente: od *CGI* skripti (pisanih u *C*-u, *C++*-u, *Shell Script*-u...), preko specijalizovanih biblioteka za jezike generalnog tipa (*Python* od verzije 1.0, *Perl* od verzije 5, *Ruby* od nacrta...), pa sve do specijalizovanih jezika i tehnologija (*PHP*, *ASP*, *ColdFusion*...). Iz statičnih Veb stranica izrodile su se dinamične Veb aplikacije, koje su korisnicima prikazivale dinamički oformljene sadržaje zasnovane na parametrima zahtjeva, praćenom ponašanju korisnika, te sigurnosnim mjerama [1]. U početku su serverske tehnologije obavljale jednostavnije zadatke (praćenje statistike o posjetiocima, distribuiranje sistema datoteka…), no vremenom je postalo moguće distribuirati čitav informacioni sistem putem Interneta, sa svim pogodnostima baza podataka i poslovne logike.

Prvu pravu poslovnu logiku koja se izvršavala na klijentskoj strani Veb aplikacija je donijela *Java Applet* tehnologija, a kasnije je sličan koncept pratio i *Macromedia Flash*. Iako spadaju u kategoriju Veb aplikacija (pa čak i SPA, u širem smislu), *Java Applet* i *Flash* aplikacije ne generišu *HTML* sadržaje na serveru i prikazuju ih na klijentovom pregledaču, već su umetnute (embedovane) u *HTML* dokument, a izvršavaju se u zasebnom procesu na klijentu, na *JVM*[12] u slučaju *Java Applet*-a, a *Macromedia Flash Player*-u u slučaju *Flash* aplikacija. Popularnost im je brzo skočila, a između ostalog su pružali mogućnosti manipulacije vektorskom i rasterskom grafikom, protok zvuka i videa, te raznovrsne interakcije sa korisnikom putem miša, tastature, kamere i mikrofona. Veliki broj reklama, igara, pomoćnih alata, pa i čitavih aplikativnih programa je razvijen u ovim tehnologijama. No, mana im je što zahtjevaju instalaciju dodatnog softvera, koji nije podržan na svim operativnim sistemima i uređajima, kao i

[11] Nije ograničen na *HTML*, može se primjeniti na bilo koji jezik označavanja.
[12] *Java Virtual Machine* - Java virtuelna mašina.

zaseban model sigurnosti. Velika prednost im je lakoća distribucije i odsutnost potrebe za instaliranjem konkretnog aplikativnog softvera na klijentskoj mašini. Ove dvije tehnologije su poprilično evoluirale do 2000. godine. *Java Applet*-i su isporučivali kompleksne aplikacije, pa čak i čitav paket *Office* programa u pregledač[13], a *Flash* je postao glavna platforma za plasiranje bogatih igara, videa i ostalih tipova multimedije u pregledaču.

Kako je Veb postao svakodnevnica, biznismeni širom svijeta su primjetili mogućnost jednostavnog, jeftinog oglašavanja, te sve više usmjeravali fokus na Veb. Ishod ovoga je bila tzv. Eksplozija Interneta[14] (eng. *Dot-com boom*), koja se desila u periodu od 1995. do 2000. Broj Veb stranica i korisnika Interneta je vrtoglavo rastao:

Godina	**Broj Veb stranica**	**Relativni rast**	**Broj korisnika Interneta**
2000.	17.087.182	438%	413.425.190
1999.	3.177.453	32%	280.866.670
1998.	2.410.067	116%	188.023.930
1997.	1.117.255	334%	120.758.310
1996.	257.601	996%	77.433.860
1995.	23.500	758%	44.838.900

Izvor: *Total Number of Websites* [6]

Investitori su "bacali novac" na sve što je vezano za Veb; u mnogo slučajeva su kompanije uzrokovale skok svojih dionica prosto dodajući "e-" prefiks ili ".com" sufiks[15]. Većina tih investitora je odlučila da previdi osnovne poslovne smjernice i umjesto toga položila velike nade u brze tehnološke napretke, u nadi da će jednoga dana dobiti povrat investicija. Na njihovu žalost, to se nije baš tako desilo; pad Eksplozije Interneta 2000. i 2001. godine je bio

[13] *Applix Vistasource Anyhere Office*
[14] Poznata još i kao *Dot-com bubble* - "eksplozija" Internet baziranih kompanija.
[15] Investiranje prefiksom, kako to naziva Majk Maznik [39].

neizbježan. No, bez obzira na strmoglav pad vrijednosti dionica i broja Internet baziranih kompanija, neki današnji giganti elektronskog poslovanja, kao *eBay*, *Amazon* i *Google*, su nastali u toj epohi i uspješno je preživjeli.

Za to vrijeme, *W3C* vodi borbu da standardizuje Veb tehnoligije (i što je bitnije, da te standarde primjene proizvođači pregledača). Izdate su verzije 2, 3, 3.2 i 4 *HTML*-a, verzija 2 *CSS*-a, te verzije 2 i 3 *ECMAScript* standarda. No, možda najznačajniji napredak po pogledu arhitekture Veb aplikacija u tom periodu donijela je tehnologija *AJAX* (*Asynchronous JavaScript and XML*), začeta 1999. Naime, Veb stanice su bile zasnovane na dobavljanju *HTML* dokumenata sa servera i prikazivanju na klijentu. Svaka interakcija sa stranicom je zahtjevala učitavanje čitavog novog dokumenta sa servera (uključujući zaglavlja, menije itd.). Ovaj proces je bio neefikasan i ogledao se lošim doživljajem za korisnika: sav sadržaj stranice nestane, sav se učita sa servera i ponovo prikaže, iako se samo neka informacija promijenila. Ovo bi uzrokovalo velikim opterećenjem za server i suvišnim protokom u mreži. U verziji 3 *Internet Explorer*-a, izdatog 1996., uveden je tag *iframe*[16] koji je dopuštao određeno asinhrono učitavanje dokumenata. Sa verzijom 5, izdatom 1999., *Internet Explorer* donosi i *ActiveX* tehnologiju, koja je kasnije široko prihvaćena i uobličena u *XMLHttpRequest JavaScript* objekat, koji dopušta slobodnu asinhronu komunikaciju klijenta sa serverom - dopremanje informacija u običnom tekstualnom ili struktuiranom (*XML* ili *JSON*) obliku klijentu, bez potrebe za učitavanjem čitavog novog dokumenta. Kao što ćemo vidjeti, *AJAX* je jedna od temeljnih tehnologija u SPA arhitekturi.

Slijedeći pad *Dot kom bum*-a, nove ideje o *ad hoc* razmjeni sadržaja, kao *Blog*-ovi[17], *RSS*[18] fidovi i *BBS*[19], su brzo uzele maha. Korisnik više nije samo pregledao sadržaje, već je mogao da ih stvara i dijeli. Ovaj novi model razmjene informacija, gdje korisnici nisu puki posmatrači, već imaju aktivnu ulogu u stvaranju sadržaja u virtuelnim zajednicama, nazvan je Veb 2.0. Pored toga, uspješno je pokrenut veliki broj *startup* kompanija orjentisanih na pružanje *online* usluga. Tako da se putem Veba mogla lako rezervisati avionska karta, *Google* je nudio

[16] *Inline frame* - okvir unutar *HTML* dokumenta koji dozvoljava prikazivanje jednog *HTML* dokumenta unutar drugog.

[17] *Blog* - skraćivanje izraza *web log* - Veb dnevnik, obično informativna ili diskusiona Veb lokacija sačinjena od pojedinačnih članaka.

[18] *Rich Site Summary* - tehnologija objavljivanja ažuriranih informacija sa Veb lokacije.

[19] *Bulletin board system* - informacioni sistem koji dozvoljava prijavljivanje korisnika, razmjenu softvera i drugih sadržaja, npr. vijesti, biltena itd., te interakciju sa drugim korisnicima, putem privatnih poruka, javnih foruma, direktnog ćaskanja itd. Preteča savremenih socijalnih mreža.

usluge pretrage Veba i lakog *online* oglašavanja, međunarodna kupoprodaja je pojednostavljana servisima kao što su *Amazon*-ova *online* robna kuća i uradi sam aukcije *eBay*-a, a *PayPal* servis je nudio usluge međunarodne razmjene novca. Veb 2.0 je uzrokovao čitav trend "verzija 2.0": *Software 2.0, Business 2.0, Learning 2.0, Health 2.0, Science 2.0* itd. - trend globalnog pružanja usluga putem Veba.

Sve ovo su podržavale napredne serverske tehnologije. U ovaj segment Veb aplikacija se uključila i *Java* sa *Servlet* i *JSP* (*JavaServer Pages*) tehnologijama. Većina popularnih serverskih tehnologija je dobila nove, dorađene verzije, neke su doživile preporod, kao *ASP* u obliku *ASP.NET*-a, a izrodilo se i nekoliko veoma popularnih razvojnih okvira (eng. framework) iz već postojećih tehnoligija, kao *Django* u *Python*-u, *Ruby on Rails* u *Ruby*-u, te *CakePHP*, *Symfony* i *Zend* u *PHP*-u.

Napredak su pratile i klijentske tehnologije. Razvijeno je dosta značajnih *JavaScript* biblioteka, kao *YUI Library*, *Dojo Toolkit*, *Ext JS* i možda najznačajnija *jQuery*, koje koriste *AJAX* i *DOM*[20] da u Veb stranice uvedu visoko interaktivno i dinamično iskustvo.

Zvanični začeci SPA koncepta nastali su 2003. u naučnom radu *Inner-browsing Extending the Browser Navigation Paradigm* [7], mada je Stjuart Moris[21] godinu ranije razvio "samoodržavajuću" Veb stranicu *slashdotslash.com*, sa sličnim konceptima. Naučni rad naglašava nedostatke klasičnog pristupa učitavanju dinamičkog sadržaja sa servera, te navodi moguće primjene klijentskih tehnologija (*IFRAME*, *JavaScript*, *XMLHTTPRequest*…) za konstrukciju Veb stranica koje nude interfejs sličniji klasičnim desktop aplikacijama - gladak, bez ponovnog učitavanja čitave stranice sa servera. 2005. Stiv Jen (*Steve Yen*) je konceptu zvanično nadjenuo ime *Single Page Application*.

Rat pregledača je nastavio da traje, mada su se učesnici mjenjali. Početkom 2003. *Apple* je objavio svoj pregledač *Safari*. Krajem 2004. iz *Netscape*-ovog projekta *Mozilla* je nastao zaseban pregledač *Firefox*. Krajem 2007. *Netscape* je zvanično objavio prestanak rada na *Navigator*-u, a sredinom 2008. se u industriju Veb pregledača uključio i *Google* sa svojim *Chrome* pregledačem.

Značajnu prekretnicu u načinu pristupa Vebu je 2007. pokrenuo *Apple* izdavanjem revolucionarnog mobilnog telefona *iPhone*. Do tada se Vebu pristupalo skoro isključivo sa

[20] *Document Object Model* - konvencija predstavljanja i interakcije sa objektima u *HTML*, *XHTML* i *XML* dokumentima.
[21] *Stuart Moris* - softver dizajner iz Velsa.

desktop i laptop računara, a sada se već predviđa da će pristupanje Vebu preko mobilnih uređaja preovladati 2015. Vrtoglav rast korisnika koji na ovaj način pristupaju Vebu je zahtjevao predefinisanje principa strukturiranja Veb stranica. Programeri su često morali praviti zasebne Veb stranice prilagođene manjim ekranima osjetljivim na dodir.

Džinovski korak ka objedinjavanju klijentskih tehnologija je uradio *HTML* sa verziom 5, praćen trećom verzijom *CSS*-a. Glavno dostignuće ovih tehnologija je nezavisnost Veb sadržaja od dodataka za pregledače[22], prvenstveno *Flash* tehnologije. Naime, pomoću *HTML5* i *CSS3* tehnologija, te pratećih *JavaScript API*-a, moguće je stvarati Veb stranice sa punom multimedijalnom podrškom bez upotrebe tehnologija treće strane. Ovo podrazumjeva animacije[23], 2D i 3D transformacije[24], tranzicije[25], protok za audio i video[26], te interaktivnu rastersku[27] i vektorsku[28] grafiku. Pored pomenutih multimedijalnih, *HTML5* specifikacija sadrži i mnoge druge korisne mogućnosti:

- *Drag and Drop* podrška;
- Raznovrsna polja za unos podataka (*URL*, broj telefona, email adresa, broj, boja, datum, vrijeme itd.);
- *Web Worker*-i koji dozvoljavaju izvršavanje skripte u zasebnoj niti;
- *Geolocation API* - praćenje lokacije uređaja;
- *Offline* skladištenje podataka pomoću *Local Storage* i *IndexedDB* tehnologija;
- Integracija *MathML*-a[29] unutar *HTML* elemenara itd.

Media upiti verzije 3 *CSS*-a nude lako rješenje za prilagođavanje izgleda Veb stranice uređaju na kom se prikazuje - standardni kompjuterski ekran, štampanje na papir, projektovanje na veliku površinu, Brajevi uređaji za slabovide, te ekrani konkretne veličine ili orjentacije.

Još jedna velika stvar koju je HTML5 postigao je značajan korak ka semantičkom Vebu. Prema riječima Berners-Lija: “Sanjam o Vebu u kom su računari sposobni da analiziraju sve

[22] *Browser plug-in* - softver koji dodaje mogućnosti u pregledač.
[23] *CSS Animations module* - http://www.w3.org/TR/css3-animations/
[24] http://www.html5rocks.com/en/tutorials/3d/css/
[25] *CSS Transitions module* - http://www.w3.org/TR/css3-transitions/
[26] *HTML5* audio i video elementi - http://www.html5rocks.com/en/tutorials/video/basics/
[27] *HTML5* canvas element - http://code.tutsplus.com/articles/21-ridiculously-impressive-html5-canvas-experiments--net-14210 i *WebGL API* - http://www.awwwards.com/22-experimental-webgl-demo-examples.html
[28] *HTML5 SVG* elementi - http://www.creativebloq.com/design/examples-svg-7112785
[29] *Mathematical Markup Language* - primjena *XML*-a za opisivanje matematičkih izraza.

podatke na Vebu - sadržaje, veze i transakcije među ljudima i računarima. "Semantički Veb", koji će ovo omogućiti, tek treba da iskrsne, ali kada se pojavi, svakodnevnim mehanizmima trgovanja, birokratije i svakodnevnog života će upravljati mašine koje međusobno pričaju. "Inteligenti agenti" koje ljudi odavno zagovaraju će se napokon materijalizovati" [8]. Podršku semantičkom Vebu *HTML5* izražava kroz čitav niz semantičkih elemenata i atributa (section, nav, article, header, footer, main, figure itd.), koji ne utiču na izgled stranice, ali dodatno opisuju namjenu njenih djelova.

U otvorenom pismu izdatom u aprilu 2010., tadašnji generalni direktor *Apple*-a Stiv Džobs[30] je oštro kritikovao *Flash* i predviđao preovladavanje otvorenih Veb standarda, kao *HTML5* i *JavaScript*, u budućnosti kako mobilnih, tako i desktop uređaja. "*Flash* više nije potreban za gledanje videa ni konzumiranje bilo kakvog sadržaja na Vebu". Među nedostacima *Flash*-a Džobs navodi veliku potrošnju energije, lošu sigurnost, nedostatak podrške za ekrane na dodir, te sam koncept dodatka za pregledač. Time je najavio da *Apple*-ovi uređaji neće podržavati *Flash* tehnologiju [9].

Veb je u procesu stalnog širena, a Veb tehnologije u stalnom razvoju. Iz međusobno povezanih tekstualnih dokumenata, *HTML* je izrastao u visoko interaktivan multimedijalni medij. Preko 30.000 PB podataka se mjesečno pošalje preko Interneta [10]. Procjenjeno je da oko 78% stanovništva razvijenih zemalja (a 40% globalno) koristi usluge Veba [11]. Skoro četvrtina čovječanstva koristi socijalne mreže [12]. Korisnici mjesečno naprave 11.944 milijarde *Google* pretraga [13]. Skoro 183 milijarde *email*-ova se pošalje dnevno [14]. Dvije milijarde minuta razgovora se dnevno obavi pomoću *Skype*-a [15]. Milijardu i 750 miliona ljudi širom svijeta koristi pametne telefone [16]. Neka od Lenisterovih predviđanja se već ostvaruju. Nadajmo se da neće sva.

[30] *Steve Jobs* (1955-2011) - američki preduzetnik i pronalazač, suosnivač firme *Apple Inc*.

3. SPA naspram ostalih Veb arhitektura

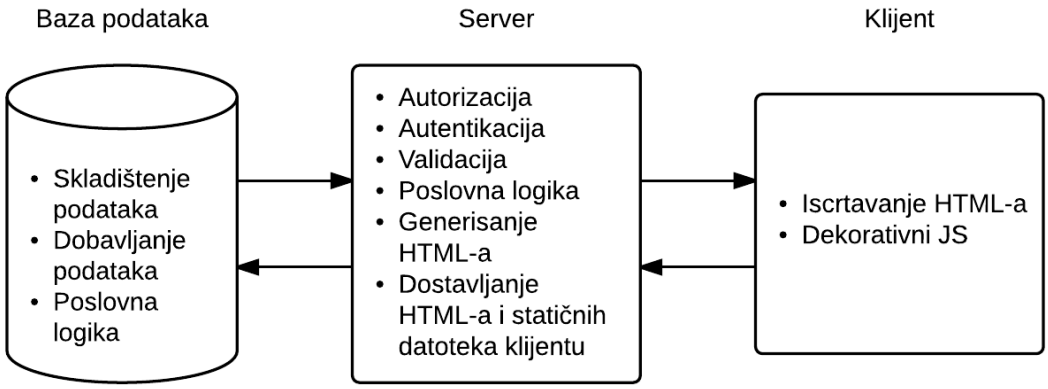
SPA arhitektura je relativno nova, ali veoma brzo hvata maha među Veb programerima. Stvoreno je nekoliko poprilično zrelih SPA razvojnih okvira, sa veoma aktivnom zajednicom koja razvija dodatke i biblioteke za pomoć u specifičnim aspektima SPA. U poglavlju koje slijedi je definisana SPA arhitektura i upoređena sa klasičnom arhitekturom Veb aplikacija i drugim aplikacijama koje se izvršavaju u pregledaču - prvenstveno *Flash* aplikacijama i *Java Applet*-ima.

3.1. Najvažnije prednosti SPA

Tradicionalna (serverska) Veb aplikacija pokušava prilagoditi paradigmu stranica povezanih linkovima razvoju aplikacija. Tako se većina popularnih serverskih razvojnih okvira fokusira na isporučivanje stranice za stranicom statičnog sadržaja. Tradicionalna Veb aplikacija u suštini radi na sljedeći način: korisnik svojom akcijom zatraži resurs (klikom na link, unošenjem *URL*-a…), pregledač na klijentu generiše zahtjev i šalje ga serveru, server generiše potpuno novu stranicu zajedno sa zaglavljem, podnožjem, menijima, tekstom, slikama, reklamama, naslovima itd., server odgovor šalje klijentu, zaslon pregledača postane bijel, pregledač obradi odgovor i iscrta novu stranicu. Ovaj proces se nastavlja tokom čitave interakcije korisnika sa aplikacijom. Tradicionalnoj Veb aplikaciji klijent služi kao korisnički interfejs - generisane Veb stranice sadrže prikaze izlaznih podataka u raznim oblicima i formulare za ulazne podatke; svu poslovnu logiku izvršava isključivo server. Postoje razni načini da se ovaj proces ubrza - optimizovanje i minimizacija *CSS*-a, kombinovanje slika u jedan *sprite*, odlaganje učitavanja *JavaScript*-a, serviranje statičnih datoteka sa *CDN*-a[31], a kada su svi resursi keširani u pregledaču, učitavanje je momentalno. No, i sa svim tim optimizacijama, pregledač pri učitavanju svake strane mora ponovo da iscrta *HTML*, te ponovo parsira i izvrši *CSS* i *JavaScript*.

[31] *Content Delivery Network* – veliki distribuirani sistem Veb servera čiji je cilj serviranje statičnih sadržaja velikom efikasnošću i s velikom dostupnošću.

Ovaj proces mnogo opterećuje server, pregledač i mrežu. Previše ima ponavljanja, a jedan od glavnih principa softverskog inžinjerstva - *DRY*[32] - se tome žestoko protivi [17]. No, ovakav pristup je prije svega istorijski veoma značajan, a sa druge strane i pogodan za Veb aplikacije određenog tipa. Na primjer, *StackOverflow*[33] - slavna *Q&A*[34] Veb lokacija za programere - funkcioniše kao tradicionalna Veb aplikacija. Njene stranice su uglavnom statične (sa manjom upotrebom *JavaScript*-a za djelimičnu interaktivnost) i sadržajno odvojene, pa svaka predstavlja zasebnu cjelinu. Dijelovi korisničkog interfejsa nemaju potrebu za međusobnom interakcijom. Skoro svaka akcija korisnika značajno mjenja stanje aplikacije, te pokreće ponovno učitavanje stranice.



Dijagram 1 – Glavna zaduženja učesnika u tradicionalnoj Veb aplikaciji

Brojne studije [18] su pokazale da i manje promjene u vremenu isporučenja stranica mogu imati mjerljiv uticaj na posjećenost Veb lokacije - što povlači znatne oscilacije u godišnjim prihodima. SPA arhitektura teži da to promjeni.

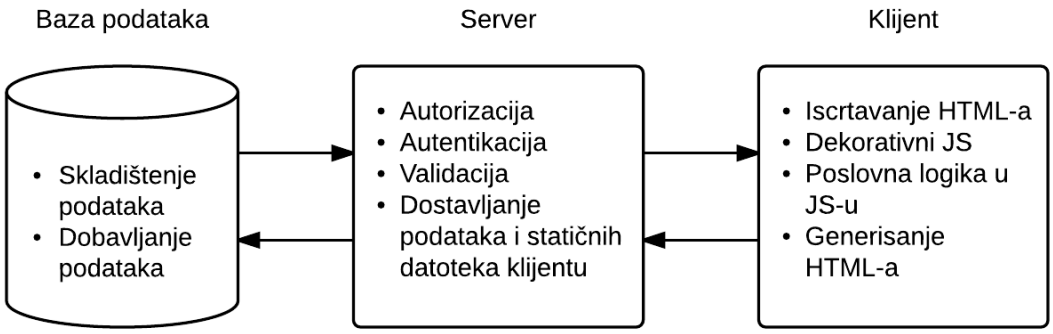
SPA je Veb aplikacija koja se dostavlja pregledaču i ne učitava se ponovo tokom upotrebe [19]. Tokom tog prvog učitavanja aplikacija učita sve potrebne resurse - *HTML, JavaScript, CSS,* podatke itd. – ili ih tokom korištenja asinhrono, pomoću *AJAX* poziva, učita i

[32] *Don't Repeat Yourself* – princip u razvoju softvera koji zagovara minimizovanje ponavljanja informacija i operacija. Obično se navodi na sljedeći način: „Svaka jedinica znanja u sistemu treba imati jedinstvenu, nedvosmislenu i mjerodavnu predstavu“.

[33] https://stackoverflow.com/

[34] *Questions and answers* - pitanja i odgovori.

ugradi u aplikaciju. U suštini, SPA je teški klijent koji se pokreće u pregledaču, distribura putem Interneta i za trajno čuvanje podataka koristi server[35].



Dijagram 2 – Glavna zaduženja učesnika u SPA

Uloge u SPA su se promjenile. Gotovo sva poslovna logika, kao i generisanje *HTML*-a su premješteni na klijenta. Ovo uzrokuje značajnim smanjenjem opterećenja kako sistemskih resursa servera, tako i protoka mreže. Klijentska strana u SPA arhitekturi više liči na standardnu desktop aplikaciju: ima svoje stanje, lokalne podatke, reaguje na događaje i veoma je odzivna, pružajući glatko i dinamično iskustvo za korisnika. Pošto se generisanje *HTML* dokumenata vrši na klijentu, ponovna iskoristivost je velika i tranzicija je veoma glatka; jednom generisano zaglavlje ne mora ponovo da se generiše i iscrtava, sve dok se sâmo ne promjeni. Server je i dalje veoma važan, samo je rasterećen obaveza. Skoro sva serverska strana je usmjerena ka radu sa podacima: podaci se trajno skladište (obično) u bazi podataka; vrši se autorizacija i autentikacija korisnika, da bi bilo jasno koji se podaci smiju poslati klijentu; nad svim podacima dospjelim s klijenta se vrši validacija, da bi se samo validni podaci trajno skladištili. Iako je obično rasterećen poslovne logike, nije potpuno isključeno u SPA da server obavlja neke zadatke koji su ipak njemu prikladniji nego klijentu, npr. generisanje datoteka ili izvršavanje algoritma kojeg vlasnik želi držati u tajnosti. Pored toga, generisanje *HTML*-a na serveru se uglanom svodi na generisanje jedne jedine stranice.

[35] Ovo je daleko najčešća upotreba SPA, no sasvim je moguće koristiti lokalnu SPA koja se u potpunosti pokreće na klijentskom računaru, a u pregledač se učitava pomoću *file URI* šeme.

SPA arhitektura je pogodna za aplikacije koje zahtjevaju visoku interaktivnost i imaju mnogo manjih stanja između kojih se zahtjeva gladak prelaz. Primjer ovakve aplikacije je *Gmail*[36], *Google*-ov Vebmejl servis. Stranice ove Veb aplikacije su dinamične i dijelovi korisničkog interfejsa su u stalnoj interakciji. Na primjer, meni sa opcijama se dinamično mijenja u odnosu na stanje liste sa mejlovima: da li su neki mejlovi označeni, da li je neki mejl otvoren za pregledanje itd. Takođe, neki dijelovi korisničkog interfejsa rade potpuno odvojeno od drugih i promjene njihovog stanja ne utiču na ostale komponente. Na primjer, odjeljak za novi mejl i njegovo stanje minimizovanosti nema potrebe da utiče na listu mejlova ili trenutno otvoreni mejl. Takođe, stanje odjeljka za ćaskanje može biti dosta kompleksno: sa kim se trenutno ćaska, koji od tih odjeljaka je minimizovan, koliko poruka je vidljivo itd. Uopšte nije pogodno na serveru pratiti ovako veliki broj manjih stanja koja se često mjenjaju, pa bi ponovno učitavanje strane na svaku akciju predstavljalo tromo iskustvo. Zato *Gmail* i jeste konstruisan kao SPA.

Prije svega, biće razjašnjen sam naziv SPA arhitektire. Šta tačno predstavlja ta "jedna strana" u „Aplikaciji u jednoj strani"? Prema Vardu Belu[37]: "(Strana je) ukupnost svega što zauzima četiri zida korisničkog interfejsa aplikacije". Zamjenom strane moglo bi se smatrati prelazak sa rasporeda Studijskog programa za matematiku i računarstvo[38] na raspored Studijskog programa za matematiku i fiziku[39]. To je poprilično razumna pretpostavka, no u SPA terminologiji to nije „strana". "Strana" u SPA je ta jedna jedina strana koju server dostavi pregledaču po pokretanju aplikacije. To je *HTML* dokument, pripremljen na serveru, koji započne aplikaciju. Nakon što se on učita, sva prezentaciona logika je na klijentu [20]. Za SPA, ponovno učitavanje je isto što i ponovno pokretanje za desktop aplikaciju. SPA koncept je dobio naziv po praksi u tradicionalnim Veb aplikacijama da server generiše strane i dostavlja ih klijentu. U SPA, server generiše samo jednu stranu, koja aplikaciju prenosi na klijenta. Učitavanje ove strane i inicijalizacija aplikacije mogu trajati duže nego učitavanje jedne strane tradicionalne Veb aplikacije, ali jednom učitani resursi se ne moraju ponovo učitavati sa servera, što čini ostatak aplikacije značajno bržim.

[36] https://mail.google.com/

[37] *Ward Bell* - potpredsjednik kompanije *IdeaBlade*, koja proizvodi softverska rješenja za efikasan prenos podataka; šest puta proglašen za *Microsoft MVP*-a.

[38] http://sekretarski-modul.apphb.com/#schedules/view?studyProgramId=1

[39] http://sekretarski-modul.apphb.com/#schedules/view?studyProgramId=2

Ukoliko se vratimo na definiciju SPA koju je dao Mikovski (aplikacija koja se dostavlja pregledaču i ne učitava više tokom korištenja), može se primjetiti da *Java Applet*-i i *Flash* aplikacije spadaju u opisanu kategoriju. Istorijski gledano, dok su *Java* i *Flash* plasirali ozbiljne aplikacije u pregledač, *JavaScript* je služio za validaciju formulara, padajuće menije i *pop-up* prozore. Ali situacija se od tada znatno promjenila, a *HTML*/*CSS*/*JavaScript* spoj ne samo da parira, već ima brojne prednosti nad *Java Applet* i *Flash* tehnologijama:

- **Ne zahtjeva dodatak za pregledač**: Korisnici pristupaju aplikaciji bez brige o instalaciji, verziji i podobnosti dodatka za pregledač sa operativnim sistemom. Takođe, programeri ne moraju da brinu o zasebnom modelu sigurnosti, koji je poznat kao problematičan kod dodataka.
- **"Lakša" aplikacija**: Aplikacija u *JavaScript*-u i *HTML*-u obično zahtjeva znatno manje sistemskih resursa, nego aplikacija kojoj je potrebno dodatno radno okruženje u zasebnom procesu.
- **Jedan klijentski jezik**: Veb programeri moraju znati dosta jezika i formata podataka - *HTML, CSS, JavaScript, JSON, XML, SQL, PHP*/*Java*/*Ruby*/*Perl*… Pošto konstrukcija iole ozbiljne Veb stranice sigurno uključuje *JavaScript* u nekoj mjeri, zašto klijentsku aplikaciju pisati u *Java*-i ili *ActionScript*-u? Korištenje jednog jezika za čitavu klijentsku aplikaciju je dobar način da joj se smanji kompleksnost.
- **Glatkija, interaktivnija stranica**: *Java* ili *Flash* aplikacija se obično prikazuje u okviru u nekom dijelu stranice i po mnogo stavki se razlikuje od ostatka *HTML* dokumenta: aplikacija se učitava tek nakon što je stranica učitana, komponente korisničkog interfejsa su drugačije, desni klik mišem je drugačiji, zvuci su drugačiji, interakcija sa ostatkom strane je minimalna. Za *JavaScript* SPA, čitav prozor pregledača je korisnički interfejs.

Sazrijevanjem, slabosti i mane *JavaScript*-a su otklonjene ili ublažene, a prednosti su dobile na vrijednosti:

- **Veb pregledač je najrasprostranjenija aplikacija na svijetu**: Veliki broj ljudi svakodnevno otvara svoj omiljeni pregledač. Pristup *JavaScript* aplikaciji je samo par klikova daleko.
- ***JavaScript* u pregledaču je jedno od najraširenijih radnih okruženja na svijetu**: Preko 2 miliona *Android* i *iOS* uređaja se aktivira dnevno [21, 22]. Svaki od njih ima

robusno okruženje za izvršavanje *JavaScript*-a ugrađeno u sam operativni sistem. Kada se tome dodaju svi desktop i laptop uređaji, dobija se zapanjujuća statistika.

- **Distribucija *JavaScript* aplikacije je trivijalna**: *JavaScript* aplikacija može postati dostupna svakom od skoro tri milijarde korisnika Veba jednostavnim hostingom na *HTTP* serveru.
- ***JavaScript* je koristan za razvijanje aplikacija koje rade na više platformi:** SPA se može konstruisati na *Windows*, *Mac OS X* ili *Linux* sistemu. Takođe, ista aplikacija se može plasirati ne samo za sve desktop, već i za tablet uređaje i pametne telefone. Ova karakteristika je rezultat konvergirajućih implementacija Veb standarda u pregledačima i zrelih biblioteka kao *jQuery*[40] i *PhoneGap*[41] koje izglađuju razlike.
- ***JavaScript* je postao iznenađujuće brz, te može, ponekad, da parira kompajliranim jezicima**: To je, opet, proizvod zahuktalog rivalstva *Mozilla*-e, *Google*-a, *Opera*-e i *Microsoft*-a na polju Veb pregledača. Savremene implementacije *JavaScript*-a uživaju pogodnosti naprednih optimizacija prevodioca, kao *JIT* kompilacija[42] na jezik mašine, *branch prediction*[43], *type-inference*[44], i višenitno izvršavanje.
- ***JavaScript* je sazrijevanjem stekao napredne mogućnosti**: Ugrađeni *JSON* objekat i *DOM* selektori slični onim iz *jQuery* biblioteke, konzistentiji *AJAX API*, *push messaging* sa *Socket.IO*[45] bibliotekom, informacije o geografskoj lokaciji klijenta sa *Geolocation API*[46], informacije o stanju klijentove konekcije sa *Network Information API*[47], vibracije uređaja sa *Vibration API*[48], stanje baterije sa *Battery Status API*[49], zauzimanje čitavog ekrana sa *Fullscreen API*[50] itd.

---

[40] http://jquery.com/
[41] http://phonegap.com/
[42] *Just-in-time compilation* - kombinacija dva tradicionalna načina prevođenja programskih jezika na mašinski kod - *ahead-of-time* kompilacije i interpretiranja - kompajliranje koda u mašinski jezik u toku izvršavanja programa.
[43] Tehnika predviđanja rezultata *if-then-else* strukture prije nego se izračuna *if* izraz.
[44] Tehnika automatskog zaključivanja rezultujućeg tipa izraza.
[45] http://socket.io/
[46] http://www.w3.org/TR/geolocation-API/
[47] http://www.w3.org/TR/netinfo-api/
[48] http://www.w3.org/TR/vibration/
[49] http://www.w3.org/TR/battery-status/
[50] http://www.w3.org/TR/fullscreen/

- ***HTML5, SVG* i *CSS3* preporuke i njihova podrška su uznapredovali**: Sada podržavaju iscrtavanje grafike detaljne do u piksel, koja parira *Java*-i i *Flash*-u po kvalitetu i brzini.
- **JavaScript se može koristiti u svim slojevima aplikacije**: Pomoću zrelih tehnologija kao što je *Node.js*[51] server čije se aplikacije pišu u *JavaScript*-u i *NoSQL*[52] baza podataka koje podatke čuvaju u *JSON* formatu. Čak je moguće koristiti iste biblioteke na serverskoj i klijentskoj strani aplikacije.
- **Desktop, laptop i mobilni uređaji su sve moćniji**: Činjenica da su procesori sa više jezgara i gigabajti *RAM*-a postali, maltene, sveprisutni znači da se obrada informacija koju je nekad morao obavljati sam server može distribuirati na klijentske sisteme.

Stoga, kada se nadalje u radu pominju SPA, misli se isključivo na one aplikacije čija je klijentska strana napisana u *HTML*/*JavaScript*/*CSS* spoju tehnologija.

JavaScript nije savršen i nije potrebno duboko kopati da bi se našli propusti, nedoslijednosti i aspekti koji bi mogli biti bolji. Ali ovo važi za sve jezike. Kada se jednom programer navikne na osnovne koncepte, primjeni najbolje prakse i nauči šta izbjegavati, razvijanje programa u *JavaScript*-u mu može postati prijatno i produktivno [19]. S obzirom na žestoko nadmetanje proizvođača pregledača i unaprijeđenja *ECMAScript* standarda, mogu se očekivati stalna poboljšanja u osobinama i performansama *JavaScript*-a. Nadolazeće verzije 6 i 7 *ECMAScript*-a obećavaju razne pogodnosti posuđene od drugih objektno orjentisanih i funkcionalnih jezika kao što su: iteratori, generatori, *array comprehension*[53], promjenljive lokalne za blok, intuitivniji lambda izrazi sa => operatorom, anotacije, moduli, klase, podrazumjevane vrijednosti argumenata, *Map*[54], *WeakMap*[55] i *Set*[56] strukture podataka itd.

Mnogi autori [23, 24, 25, 26, 27, 28, 20] u SPA arhitekturi vide budućnost Veba. Mnoge kompanije prelaze na SPA model na svojim Veb lokacijama; *Facebook*, *Twitter*, *Pinterest*,

[51] http://nodejs.org/

[52] *Not Only SQL* - baze podataka koje smještaju podatke u obliku drugačijem nego SQL, koji je zasnovan na međusobno povezanim tabelama.

[53] Sintaksna konstrukcija kojom se formira niz vrijednosti iz već postojećih nizova, veoma slično matematičkoj notaciji za definisanje skupa navođenjem osobina njegovih elemenata.

[54] Kolekcija parova (ključ, vrijednost), gdje i ključ i vrijednost mogu biti bilo kog tipa.

[55] Kolekcija parova (ključ, vrijednost), gdje je ključ objekat, a vrijednost proizvoljnog tipa.

[56] Kolekcija koja čuva jedinstvene vrijednosti bilo kog tipa.

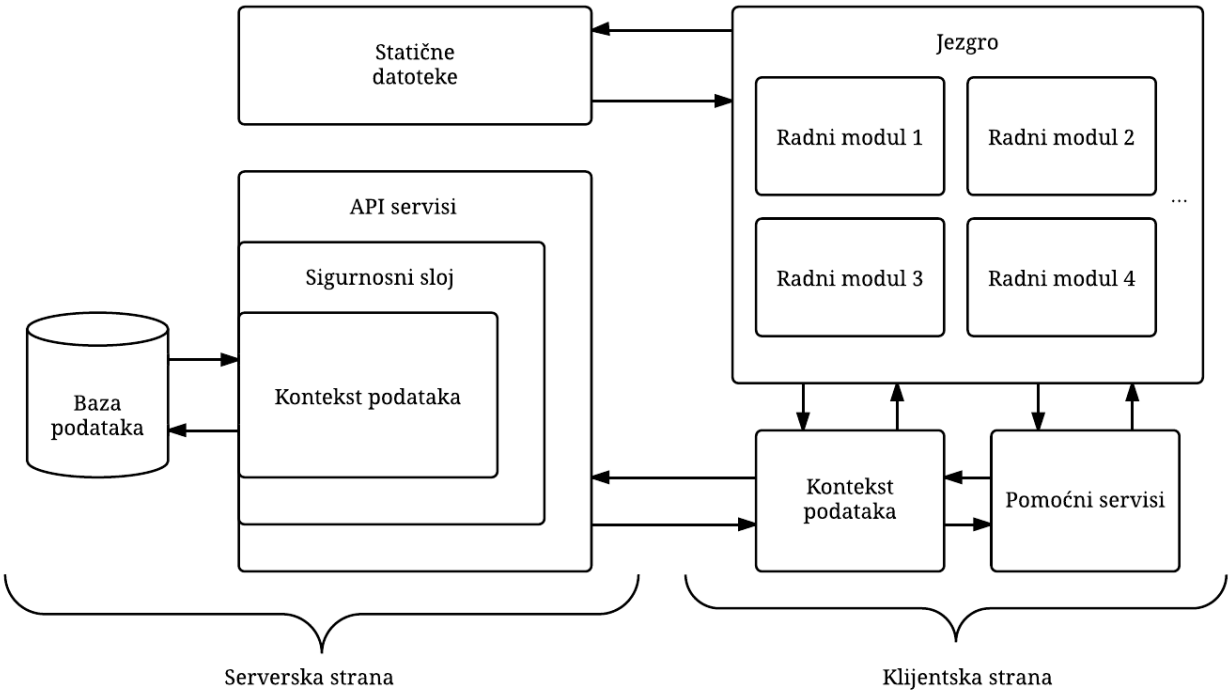
Google i *Microsoft* su samo neki primjeri. Programeri širom svijeta prepoznaju prednosti SPA arhitekture, kojoj se predviđa svijetla budućnost u sferi razvoja softvera.

3.2. Sažet prikaz organizacije SPA

SPA se sastoji od dvije sasvim odvojene aplikacije - klijentske i serverske. Klijentska aplikacija implementira većinu poslovne logike i svu komunikaciju sa korisnikom. Serverska aplikacija uglavnom služi kao omotač za sloj podataka i implementaciju sigurnosti. Sem podataka, server klijeta opslužuje i statičnim datotekama (*HTML, JavaScript, CSS*), koje klijent traži po potrebi. Sva komunikacija između serverske i klijentske aplikacije se vrši asinhrono.

Naglasak organizacije klijentske strane je na modularnosti. Modularan kôd je kôd razdvojen u nezavisne cjeline sa jasno odvojenom odgovornošću. Modularnost čini dostavljanje dijelova aplikacije pregledaču lakšim, a sam kôd razumljivijim, lakšim za testiranje, održavanje i mjenjanje [29]. Neophodan je jedan glavni modul, obično nazivan ljuskom (eng. *shell*), koji će se starati za životni ciklus klijentske aplikacije i interakciju sa pregledačem. Ljuska usklađuje funkcionisanje radnih modula, pomoćnih modula i univerzalnih interfejsa pregledača, kao što su *URL*, istorija i kolačići. Takođe je odgovorna za generisanje *HTML*-a, upravljanje stanjem aplikacije i komunikaciju među radnim modulima. Radni moduli obavljaju konkretnu poslovnu logiku i rade nezavisno koliko god je to moguće [19]. Pomoćni servisi čuvaju podatke i pružaju pomoćne usluge, koje su često potrebne u više različitih radnih modula. Među njima se izdvaja "kontekst podataka" na klijentskoj strani - servis za dobavljanje podataka sa servera, njihovo keširanje na klijentu, te proslijeđivanje radnim modulima po zahtjevu.

Serverska aplikacija se sastoji od statičnih datoteka i slojevitog sistema za pristup podacima. Obično se organizuje kao skup *API* servisa kojima pristupa kontekst podataka klijenta. *API* poziv, koji uspješno prođe sigurnosne provjere, pristupa kontekstu podataka na serveru - servisu koji komunicira sa bazom podataka - i klijentu se šalje odgovor u *JSON* formatu. Kontekst podataka na serveru pribavlja podatke iz baze i priprema ih za slanje klijentu ili obrađuje i validuje podatke sa klijenta i smješta ih u bazu.



Dijagram 3 – Organizacija SPA

3.3. Opis sistema za rukovanje rasporedima nastave

Kao što je u uvodu naznačeno, kroz rad će se kao primjer često koristiti aplikacija (SPA, naravno) za rukovanje rasporedima nastave, nazvana "Sekretarski modul"[57]. Ovako je nazvana jer joj je glavna namjena da pomogne sekretarima studijskih programa Filozofskog Fakulteta Univerziteta u Istočnom Sarajevu pri konstrukciji i objavljivanju rasporeda nastave. Pored toga, sistem povezuje studente, dekanat i skriptarnicu, te generiše izvještaje vezane za nastavu.

Klijentska aplikacija je konstruisana u *DurandalJS JavaScript* SPA razvojnom okviru, koji koristi *KnockoutJS* za dinamičko generisanje *HTML*-a. Kontekst podataka na klijentu je konstruisan pomoću *BreezeJS*, a modularizacija postignuta pomoću *RequireJS* biblioteke.

Serverska aplikacija je konstruisana u *ASP.NET* razvojnom okviru. *API* servisi su konstruisani kao mješavina *ASP.NET MVC* i *WebAPI* poziva. Kontekst podataka na serveru je konstruisan korištenjem *Entity Framework*-a, pomoću *Code First* pristupa, i ima mogućnost

[57] http://sekretarski-modul.apphb.com/

komunikacije sa *SQL Server* bazom podataka. Za autentikaciju i autorizaciju je iskorišten *CodeFirst Membership Provider*.

4. Server kao API

Dok korisnik krstari tradicionalnom Veb lokacijom[58], server troši dosta procesorske moći na generisanje i slanje stranice za stranicom sadržaja. U SPA, server je potpuno drugačiji. Gotovo sva poslovna, a čitava prezentaciona logika je premještena na klijeta [19]. Server uopšte ne prati stanje korisničkog interfejsa. U stvari, server uopšte ne odlučuje šta aplikacija radi [20]. Uloga servera je, i dalje, nezamjenjiva, ali je "tanji" i usredsrjeđen na pružanje usluga kao što su trajno čuvanje i validacija podataka, autentikacija i autorizacija korisnika, sinhronizacija podataka [19], te serviranje statičnih datoteka po zahtjevu. Tako se serverska logika obično organizuje kao *API* - skup javnih funkcionalnosti koje pružaju usluge interakcije sa sistemom. Ovaj pristup čini odvojenost serverske i klijentske strane jasnom, a sistem ostavlja dostupnim za komunikaciju sa različitim klijentskim aplikacijama, ne samo jednom SPA.

Iako klijentska strana upravlja svojim lokalnim podacima, za neke aspekte aplikacije je ipak podobniji server. Tu spada i sigurnost. Poznata praksa u konstrukciji klijent/server aplikacija je da se ne vjeruje ničemu što stiže s klijenta. Ovo je opravdano iz više razloga:

- Klijentska aplikacija se u potpunosti izvršava na klijentskom sistemu. Ovo zahteva distribuciju koda čitave aplikacije klijentu, što ga čini podložnim zlonamjernom obrnutom inžinjeringu. Zlonamjerni korisnik može promjeniti kôd aplikacije, obilazeći sigurnosne provjere.
- Podaci na klijentskoj strani su privremeni (dostupni samo dok korisnik koristi aplikaciju) i podložni neželjenoj ili zlonamjernoj promjeni.
- Ukoliko je server organizovan kao *API*, može mu se pristupati sa raznih klijentskih aplikacija. Svaka od njih može imati različit bezbjednosni sistem, što komplikuje sistem i čini ga ranjivijim.
- Zahtjevi ne moraju uopšte dolaziti iz klijentskog djela SPA ili Veb aplikacije uopšte.
- Zahtjev klijenta se može presresti ili izmjeniti kao rezltat greške u komunikacionom kanalu.

[58] Veb lokacijom koje je konstruisana koristeći tradicionalne, serverske tehnologije.

4.1. Autentikacija i autorizacija

Autentikacija je proces utvrđivanja da je neko stvarno osoba (računar) kojom se predstavlja. Obično započinje kada korisnik unese korisničko ime i lozinku [19]. U ovom pogledu je server neophodan zbog pomenute prijetnje obrnutim inžinjeringom. Ukoliko bi se za autentikaciju oslonili isključivo na klijentsku aplikaciju, koliko god primjenjivali minimizaciju[59] i obfuskaciju[60], nikada ne možemo biti sigurni da zlonamjerni korisnik neće uspjeti modifikovati autentikacionu logiku i zaobići sisteme sigurnosti, pretvarajući se da je neko drugi. Stoga je neophodno autentikacionu logiku smjestiti na server, gdje će biti sigurna od pokušaja obrnutog inžinjeringa.

Autorizacija je proces osiguravanja da samo oni ljudi (računari) koji smiju da pristupe informacijama i operacijama iste i mogu koristiti. Ovo se obično postiže dodjeljivanjem ovlaštenja ili uloga korisnicima; kada je korisnik prijavljen na sistem, jasno se zna kom dijelu sistema smije da pristupi. Veoma je bitno da se autorizacija primjeni na serverskoj strani. U suprotnom, iako klijentska aplikacija ne prikaže nedozvoljene podatke korisniku, on im veoma lako može pristupiti čak i bez potrebe za obrnutim inžinjeringom, kada su već prenijeti na klijentski sistem. Povoljna nuspojava autorizacije je da se količina podataka poslatih klijentu minimalizuje, s obzirom da se šalju samo podaci za koje korisnik ima ovlaštenja, čineći transakciju potencijalno bržom [19].

Programeri se sve više oslanjaju na spoljne autentikacione servise, npr. one koje pružaju *Facebook*, *Twitter* i *Google*. Neka je na primjer u konstruisani sistem integrisana podrška za neki spoljni servis *OAuth*[61] autentikacije. Neophodno je kod korištenog servisa registrovati konstruisani sistem kao klijenta. Pri tome se od servisa dobija jedinstven javni identifikator i tajni ključ. Pri prijavi na konstruisani sistem, korisnik se usmjerava na stranicu za prijavu korištenog servisa. Nakon uspješne prijave na servis i odobravanja konstruisanom sistemu da koristi korisnikove podatke sa servisa, servis konstruisanom sistemu šalje autorizacioni kôd. Konstruisani sistem koristi prispjeli kôd da od servisa zatraži pristupni token. Pri tome

[59] *Minification* – Uklanjanje nepotrenih karaktera iz kôda, bez promjene funkcionalnosti. Obično se uklanjaju karakteri za bijeli prostor i novi red, te nepotrebne oznake za blok, a varijable se preimenuju u kraće nazive.
[60] *Obfuscation* – Namjerno pisanje (ili generisanje) koda nečitkog ljudima, sa namjerom da mu se sakrije namjena ili oteža obrnuti inžinjering.
[61] Otvoreni standard za autentikaciju. Pruža siguran način da klijentske aplikacije pristupe zaštićenim resursima na serveru u ime njihovih vlasnika.

konstruisani sistem koristi svoj identifikator i tajni ključ. Pomoću pristupnog tokena, konstruisana aplikacija može pristupiti korisnikovim podacima sa korištenog servisa. Obično servisi implementiraju i neki sistem ograničenja pristupa, pa korisnik može eksplicitno aplikaciji dozvoliti ili opovrgnuti pristup određenim podacima.

Na ovaj način, konstruisani sistem treba čuvati svoj identifikator za spoljašnji servis i svoj tajni ključ, a od podataka o korisniku samo pristupni token. U zavisnosti od korištenog servisa, moguće je da je token potrebno osvježavati. Sa druge strane, korisnik ne mora da pamti još jedno korisničko ime i lozinku.

4.2. Validacija

Validacija je proces kontrole kvaliteta podataka, osiguravanje da se isključivo precizni i razumni podaci trajno čuvaju. Validacija pomaže u spriječavanju nastanka grešaka i njihovog širenja na druge sisteme [19]. Na primjer, sistem za upravljanje rasporedima osigurava da korisnik konstruiše raspored nastave za dan u budućnosti, a prije kraja tekućeg semestra. Bez te provjere bi korisnik mogao promjeniti raspored već održane nastave (te tako uticati na izvještaje) ili zakazati nastavu van školske godine.

Važno je primjeniti validaciju i na klijentskoj i na serverskoj strani: na klijentskoj strani kao način brzog obavještavanja korisnika o greškama, a na serverskoj strani prije trajnog čuvanja, jer se nikad ne može vjerovati da su podaci pristigli sa klijenta validni. Razni problemi mogu uzrokovati da server dobije podatke s greškom:

- Greška u programu može onesposobiti klijentsku validaciju;
- Neka druga klijentska aplikacija možda ne sadrži validaciju - Veb server aplikacijama često pristupa više različitih klijentskih aplikacija;
- Mogućnost koja je validna u trenutku slanja ne mora biti validna u trenutku dolaska na server (recimo, sjedište u avionu je rezervisano nakon što klijent klikne na “Rezerviši”);
- Zlonamjerni korisnik može pokušati probiti sigurnosni sistem ili onesposobiti aplikaciju ubacivanjem nevalidnih podataka u sistem; recimo napadima umetanjem *SQL* upita (*SQL injection*) ili prekoračenjem bafera (*Buffer overflow*) [19].

4.3. Trajno čuvanje i sinhronizacija podataka

Iako SPA može da čuva podatke na klijentskoj strani, ti podaci su privremeni i lako se mogu promjeniti van kontrole aplikacije; recimo od strane pregledača ili njegovih dodataka, operativnog sistema itd. U većini slučajeva, na klijentu se čuvaju podaci trenutno potrebni za rad aplikacije, dok se trajni podaci čuvaju na serveru, najčešće u bazi podataka.

Podaci se takođe nekad moraju sinhronizovati među više klijenata. Recimo, aplikacija *Google Docs* dozvoljava kolaborativnu obradu tekstualnih dokumenata. To podrazumjeva da se lokacija kursora, kao i promjene nad dokumentom svakog korisnika djele sa ostalim učesnicima. Ovo se obično postiže tako što klijentska aplikacija trenutno stanje pošalje serveru, koji ga sačuva i objavi svim klijentima u kolaboraciji. Sinhronizacija se takođe koristi sa prolaznim podacima; na primjer, kada se koristi server za ćaskanje koji prosljeđuje poruke među klijentima: iako server ne čuva poruke, ključna je njegova uloga proslijeđivanja poruka odgovarajućim autentikovanim klijentima [19].

4.4. Serverske tehnologije u SPA

Klijentska strana modernih SPA se konstruiše isključivo u tehnologijama ugrađenim u pregledač - *HTML*, *CSS*, *JavaScript*, *SVG* itd. Nasuprot tome, serverska strana se može organizovati u bilo kojoj serverskoj tehnologiji. Klijentska aplikacija je apsolutno "agnostična na server"; potrebna joj je samo serverska aplikacija koja će je opsluživati statičnim datotekama i podacima. S obzirom da postoje mnoge serverske tehnologije, različiti faktori mogu uticati na izbor odgovarajuće:

- **Poznavanje tehnologije**: S obzirom da skoro svaka serverska tehnologija može poslužiti pri izradi SPA, poznavanje jezika ili razvojnog okvira može značajno smanjiti trajanje konstrukcije.
- **Zrelost biblioteka i razvojnih okvira**: Ovde se prije svega misli na biblioteke specijalizovane za komunikaciju sa bazom podataka, sisteme autentikacije i autorizacije i

druge pomoćne biblioteke. Ukoliko jezik ima robusnu *ORM*[62] i autentikacionu biblioteku, vjerovatno je veoma pogodan za serversku stranu SPA.

- **Prilagođenost *REST* arhitekturi**: Pošto se server u SPA uglavnom organizuje kao *API*, u kom većina servisa obavlja *CRUD*[63] operacije, veoma je pogodan za oblikovanje kao *REST*[64] servis. Ovim se postiže jednostavniji razvoj i održavanje, kao i jednostavnija komunikacija između klijentske i serverske aplikacije. Mnoge serverske tehnologije imaju specijalizovane *REST* biblioteke.
- **Prilagođenost zahtjevima SPA**: Iako svaka serverska tehnologija *može* poslužiti za serversku stranu SPA, neke više odgovaraju za ulogu koju server igra u SPA. *Apache*[65] server, na primjer, briljira u brzom generisanju pojedinačnih *HTML* dokumenata i njihovom dostavljanju klijentu. On održava niz procesa (ili niti) koji čekaju da usluže dolazeće zahtjeve. Ukoliko su svi servisni procesi (niti) zauzeti, novi zahtjev čeka da se jedan servisni proces (nit) oslobodi. Stoga, *Apache* stavlja naglasak na što brže generisanje odgovora i oslobađanje procesa (niti). Ovo je poželjno za tradicionalnu Veb aplikaciju, ali ne uvijek i za SPA. Moderne SPA često zahtjevaju čestu razmjenu poruka u skoro realnom vremenu sa serverom, što zahtjeva potencijalno desetine hiljada konekcija koje ostaju otvorene. Neprekidna konekcija se može oponašati *long-polling* tehnikom, ali to u *Apache* serveru blokira servisni proces (nit) pa tako i nadolazeće zahtjeve, a koristi i značajne sistemske resurse. Sa druge strane, *Node.js* server je neblokirajući server čije su aplikacije zasnovane na događajima. On sadrži jedan glavni red događaja (*Event queue*), a svaki zahtjev izaziva jedan ili više događaja, koji se konkurentno obrađuju. Ukratko, jedna instanca *Node.js* servera na skromnom hadrveru može opslužiti stotine hiljada konkurentnih otvorenih konekcija, kakve su nekad potrebne za SPA. Mnogi testovi [30, 31, 32], među kojima je možda najznačajniji *PayPal*-ov [33], pokazuju značajno bolje performanse *Node.js* servera nad tradicionalnijim serverima, kao

[62] Objektno-relaciono mapiranje - tehnika konvertovanja podataka između relacionog modela, čestog u bazama podataka, i objektnog modela, u kom se podaci predstavljaju u objektno-orjentisanim jezicima.
[63] *Create*, *Read*, *Update*, *Delete* - pravi, čitaj, mjenjaj, briši. Osnovne operacije za obradu podataka.
[64] *Representational State Transfer* - stil oblikovanja mrežnih aplikacija u kojem se podaci i funkcionalnosti posmatraju kao resursi. Resursima se pristupa pomoću *URI*-ja, a upravlja korištenjem predefinisanog skupa jednostavnih, jasno definisanih operacija.
[65] http://www.apache.org/

što su *Apache* ili *Tomcat*, u situacijama kada je neophodno obraditi veliki broj zahtjeva istovremeno.

- ***JavaScript* „s kraja na kraj“**: Za konstrukciju Veb aplikacije je potrebno znati dosta jezika: za pristup bazi podataka obično neki dijalekt *SQL*-a, za serversku obradu jedan od mnogih serverskih jezika (*PHP*, *Java*, *Python*, *Ruby*, *C#*...), *JavaScript* za poslovnu logiku na klijentskoj strani, *HTML* za strukturu i *CSS* za stil stranice. Često prelaženje s jednog jezika na drugi brzo postaje zamorno i čest je izvor grešaka. Takođe, utroši se dosta procesorskog vremena u konvertovanju iz jednog formata podataka u drugi (relacioni iz baze u objektni na serverskoj strani u *JSON* na klijentskoj strani i nazad u drugom smjeru). Ovo je dodatna prednost za *Node.js*, kao i *NoSQL* baze podataka koje podatke čuvaju u *JSON* obliku. Korištenjem ovih tehnologija moguće je baš svu poslovnu logiku, kao i pristup bazi, pisati na jednom jeziku, a uklonjena je i potreba za konverzijom formata podataka. Čak je moguće da se iste bibiloteke koriste sa obje strane, kao recimo validaciona logika.

5. Primjer implementacije serverske aplikacije u *ASP.NET* tehnologiji

Pri izboru serverskih tehnologija korištenih u Sekretarskom modulu, autor se najprije vodio prvim dvijema stavkama navedenim u prošlom potpoglavlju, te odsustvom potrebe za komunikacijom u realnom vremenu klijenta sa serverom.

ASP.NET je zreo razvojni okvir za konstrukciju Veb aplikacija, podržan robusnim bibliotekama iz *.NET* pakētā. *Entity Framework* (*EF* u daljem dijelu teksta), *ORM* biblioteka iz *ADO.NET* paketa, pruža mnoge pogodnosti u modelovanju i pristupu bazi podataka, uglavnom bez ikakve potrebe za pisanjem *SQL*-a. *System.Web.Security* prostor imena sadrži klase za implementaciju autentikacije i autorizacije u *ASP.NET* aplikaciji, zasnovane na čvrstim principima kriptografije i godinama usavršavanja. Kao jedan od glavnih nedostataka *ASP.NET*-a se često navodi činjenica da je neophodan (*closed source*) *Windows Server* za pokretanje, ali *open source* projekat *MONO*[66] je to premostio. Koristeći *MONO*, *C#* aplikacije se mogu pokrenuti i na *Linux* i *Mac* baziranim operativnim sistemima, uključujući i *Android* i *iOS* platforme.

U sljedećem potpoglavlju će biti prikazan okvirni primjer konstrukcije serverske aplikacije u *ASP.NET* razvojnom okviru.

5.1. Kontekst podataka i sigurnosni sistem

Najprije je potrebno konstruisati kontekst podataka. *EF* podržava tri pristupa modeliranju sistema za komunikaciju sa bazom podataka:

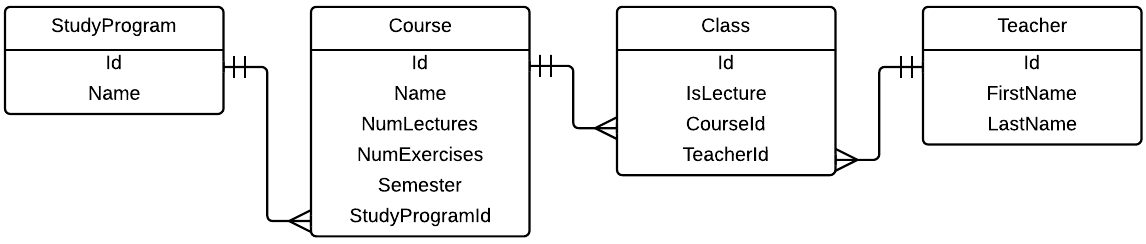
- *Database First* - Klasični pristup konstrukciji programa koji komuniciraju sa bazom podataka, sa dodatnim pogodnostima. Baza podataka se konstruiše odvojeno, a *EF* generiše klase za pristup bazi. Ovim pristupom imamo više kontrole nad bazom i njenim naprednim mogućnostima - transakcije, uskladištene procedure itd. - dok se na generisani kôd ne može puno uticati.
- *Code First* - Pristup koji naglasak stavlja na kôd za pristup bazi. Programer konstruiše klase koje predstavljaju model podataka, a *EF* generiše bazu.

[66] www.mono-project.com/

- *Model First* - Najapstraktniji pristup. Koristeći *Entity Framework Designer* programer modeluje podatke pomoću *UML* dijagrama iz kojih *EF* generiše bazu i klase za pristup bazi.

U radu će biti prikazan drugi pristup – *Code First* – pošto se u serverskom dijelu SPA želi što više kontrole nad načinom pristupa podacima. Takođe, SPA arhitektura obično ne sadrži poslovnu logiku na nivou baze, pa uskladištene procedure nisu poželjne. Kontekst podataka na serveru će se sastojati iz dvije cjeline: definicije konteksta podataka (definicije tabela, pomoću *POCO*[67] modela i *DbSet* objekata) i inicijalizatora konteksta podataka.

Neka se podaci sistema sastoje od predmeta (*Course*), od kojih svaki sadrži podatke o nazivu, nastavnom fondu (sedmični broj časova), semestru u kom se održava i studijskom programu kome pripada. Studijski program (*StudyProgram*) je karakterisan nazivom i kolekcijom predmeta. Nastavnika (*Teacher*) karakterišu ime i prezime. Između predmeta i nastavnika postoji "n na m" veza koja se ostvaruje preko pokrivenosti nastave (*Class*). Pored predmeta i nastavnika koje povezuje, svaka stavka pokrivenosti sadrži podatak od tome da li je su u pitanju vježbe ili predavanja.



Dijagram 4 – ERM baze

Od ovog modela potrebno je napraviti *POCO* klase. *EF* koristi niz anotacija kojima se dodatno opisuju osobine ciljnih kolona tabela u bazi, te neke opcije automatske validacije. Sve

[67] *Plain Old Class Objects* - obični objekti klasa - podrazumjeva korištenje klasa koje ne naslijeđuju neku klasu koju definiše razvojni okvir.

ove anotacije se nalaze u *System.ComponentModel.DataAnnotations* prostoru imena, a više o njima se može naći na Vebu[68].

Model studijskih programa može izgledati ovako:

```
public class StudyProgram
{
    public int Id { get; set; }
    [Required, MinLength(5), MaxLength(50)]
    public string Name { get; set; }
}
```

Nakon što *EF* generiše bazu, tabela u kojoj su smješteni studijski programi će se zvati *StudyPrograms* i imati kolone *Id* i *Naziv*.

EF koristi princip "konvencije prije konfguracije". To znači da pri slikanju modela u tabele baze *EF* koristi niz podrazumjevanih i najčešće korištenih postavki, koje se mogu lako po potrebi izmjeniti. Na primjer, naziv ciljne tabele je naziv klase modela u obliku množine na engleskom jeziku. Takođe, atribut *Id* se slika u kolonu pod nazivom *Id*, koja će primarni ključ tabele. U *EF* konvencija je da atribut cjelobrojnog tipa sa nazivom *Id* ili *NazivKlaseId* predstavlja primarni ključ ciljne tabele, sa *IDENTITY(1, 1)* svojstvom.

EF razumije i tipove atributa - kolona *Id* će biti tipa *INT*, a *Name* tipa *VARCHAR(max)*. Pored toga, kolona *Name* će biti obavezna, a nad njom će se vršiti automatska validacija po dužini - mora sadržati između 5 i 50 karaktera - sve zahvaljujući anotacijama.

Sve navedene konvencije se mogu prekonfigurisati - naziv tabele sa *Table* anotacijom nad klasom, naziv kolone sa *Column* anotacijom nad atributom, da li će se atribut uopšte slikati u kolonu sa *NotMapped* anotacijom, primarni ključ drugačijeg naziva se označava *Key* anotacijom itd.

Model predmētā, koji je na "n" kraju veze sa studijskim programima, može da izgleda ovako:

[68] Na primjer, na zvaničnoj *Microsoft*-ovoj Veb lokaciji na adresi: http://msdn.microsoft.com/en-us/data/gg193958.aspx.

```
public class Course
{
    [Key]
    public int Id { get; set; }
    [Required, MaxLength(100)]
    public string Name { get; set; }
    [Required, Range(0, 100)]
    public int NumLectures { get; set; }
    [Required, Range(0, 100)]
    public int NumExercises { get; set; }
    [Required, Range(0, 8)]
    public int Semester { get; set; }
    [Required]
    public int StudyProgramId { get; set; }
    public StudyProgram StudyProgram { get; set; }
}
```

Može se primjetiti atribut *StudyProgramId*, očekivan strani ključ. Dovoljno je navesti njega i tzv. navigacioni atribut *StudyProgram*, i *EF* će uspostavi "1 na n" vezu između dvije odgovarajuće tabele. Opet, u pitanju je konvencija: strani ključ nosi naziv *CiljnaTabelaId*, a navigacioni atribut *CiljnaTabela*. Navigacioni atribut je neophodan da bi *EF* „znao" koja je tabela sa "1" kraja veze, a u programu predstavlja zgodnu olakšicu:

```
StudyProgram studyProgram = course.StudyProgram;
```

Sa druge strane, u klasi *StudyProgram* se takođe može dodati navigaciono svojstvo. Ovo može biti zgodno ukoliko je u programu potrebno brzo pristupiti svim predmetima studijskog programa. Pošto jedan studijski program može imati više predmeta, atribut mora biti kolekcija:

```
public virtual ICollection<Course> Courses { get; set; }
```

Naziv kolekcije treba biti u obliku množine na engleskom jeziku ciljnog entiteta, po konvenciji. Samo jedan od ova dva navigaciona atributa je neophodan da bi *EF* uspješno uspostavio vezu. Ključna riječ *virtual* omogućuje tzv. lijeno učitavanje (eng. *lazy loading*) iz baze - pri učitavanju studijskog programa iz baze se ne učitavaju svi njegovi predmeti, već se moraju eksplicitno tražiti. Ovo omogućuje minimalizovanje količine učitanih podataka.

Konvencija se, naravno, može prekonfigurisati. Pomoću *ForeignKey* i *InverseProperty* anotacija veza se može konfigurisati po volji. Naime, navigaciono svojstvo sa "n" strane veze se

notira sa ove dvije anotacije, *ForeignKey* anotaciji se proslijedi naziv atributa koji predstavlja strani ključ, a *InverseProperty* anotaciji se proslijedi naziv navigacionog svojstva sa "1" strane veze. Dopunjene klase *StudyProgram* i *Course* mogu da izgledaju ovako:

```
public class StudyProgram
{
    [Key]
    public int Id { get; set; }
    [Required, MinLength(5), MaxLength(50)]
    public string Name { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}
```

```
public class Course
{
    [Key]
    public int Id { get; set; }
    [Required, MaxLength(100)]
    public string Name { get; set; }
    [Required, Range(0, 100)]
    public int NumLectures { get; set; }
    [Required, Range(0, 100)]
    public int NumExercises { get; set; }
    [Required, Range(0, 8)]
    public int Semester { get; set; }
    [Required]
    public int StudyProgramId { get; set; }
    [ForeignKey("StudyProgramId"), InverseProperty("Courses")]
    public StudyProgram StudyProgram { get; set; }
    public ICollection<Class> Classes { get; set; }
}
```

Po uzoru na prethodne klase, može da se konstruiše ostatak modela:

```
public class Teacher
{
    [Key]
    public int Id { get; set; }
    [Required, MaxLength(20)]
    public string FirstName { get; set; }
    [Required, MaxLength(20)]
    public string LastName { get; set; }
    public ICollection<Class> Classes { get; set; }
}
```

```
public class Class
{
    [Key]
    public int Id { get; set; }
    [Required]
    public bool IsLecture { get; set; }
    [Required]
    public int CourseId { get; set; }
    [ForeignKey("CourseId"), InverseProperty("Classes")]
    public virtual Course Course { get; set; }
    [Required]
    public int TeacherId { get; set; }
    [ForeignKey("TeacherId"), InverseProperty("Classes")]
    public virtual Teacher Teacher { get; set; }
}
```

Koristeći definisane modele, konstruiše se i sam kontekst podataka:

```
public class DataContext : DbContext
{
    public DbSet<Course> Courses { get; set; }
    public DbSet<StudyProgram> StudyPrograms { get; set; }
    public DbSet<Teacher> Teachers { get; set; }
    public DbSet<Class> Classes { get; set; }
}
```

Klasa koja predstavlja kontekst podataka treba da nasljeđuje *DbContext* klasu iz *System.Data.Entity* prostora imena. Za svaku tabelu u bazi neophodno je napraviti javni atribut tipa *DbSet<T>*, gdje *T* predstavlja klasu modela.

Nakon pokretanja programa, *EF* će generisati bazu. Ovaj proces je, opet, pod uticajem konvencije. *EF* generiše bazu pod nazivom prve klase koja nasljeđuje *DbContext* koja se nalazi u projektu. Takođe, *EF* će napraviti *SQL Server Express* bazu ukoliko je to moguće, a *LocalDB* bazu ukoliko nije i to u *App_Data* poddirektorijumu projekta. Sve ovo se može lako prekonfigurisati. U slučaju *ASP.NET* aplikacije, konfigurisanje se može izvršiti u *Web.config XML* datoteci, dodavanjem *connectionStrings* taga u korjeni *configuration* tag:

```
<connectionStrings>
    <add name="OurDataBase"
        providerName="System.Data.SqlClient"
        connectionString="Data Source=(LocalDB)\v11.0;
            AttachDbFileName=|DataDirectory|\DataContext.mdf;
            Integrated Security=True" />
</connectionStrings>
```

Atribut *name* predstavlja jedinstveno ime konekcije programa na bazu. Atribut *providerName* predstavlja puni naziv biblioteke koja vrši krajnju komunikaciju sa bazom. Pored biblioteka za Microsoft-ove baze podataka (*SQL Server* u raznim oblicima, *SQL Compact Edition* i *LocalDB*) koje se isporučuju sa *.NET* paketom, postoje zvanične biblioteke za komunikaciju sa *MySQL*, *Oracle*, *SQLite*, *PostgreSQL* i druge tipove baza podataka. Atribut *connectionString* služi za određivanje tipa, konkretne lokacije baze, te brojnih drugih postavki ciljne baze. Više informacija o ovoj temi se može pronaći na Vebu[69]. Očigledno, većina posla se obavlja "pod haubom", no to je upravo prednost zrele biblioteke kao što je *Entity Framework*. Pruža veliki stepen apstrakcije, dok većinu repetativnog posla obavlja sama.

Pored anotacija, *EF* nudi i *fluent API* - elokventan način konfigurisanja modela. Anotacije pokrivaju samo podskup funkcionalnosti *fluent API*, pa postoje scenariji koje anotacije ne pokrivaju. *Fluent API* se obično koristi predefinisanoj metodi *OnModelCreating*, naslijeđenoj iz klase *DbContext*. Tu se mogu detaljnije opisati slikanja atributa u kolone, klasa u tabele, veze među tablama, korištene konvencije itd.:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
      // Ukanja konvenciju imenovanja tabla kao množine naziva klase modela
      modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

      // Definiše kompozitni primarni ključ na tabeli Class
      modelBuilder.Entity<Class>()
            .HasKey(c => new { c.CourseId, c.TeacherId });

      // Konfiguriše vezu "1 na n" između entiteta StudyProgram
      // i Course gdje se redovi na "n" strani kaskadno brišu,
      // a strani ključ je obavezan
      modelBuilder.Entity<Course>()
            .HasRequired(c => c.StudyProgram)
            .WithMany(sp => sp.Courses)
            .WillCascadeOnDelete(true);
}
```

Prije konstrukcije inicijalizatora konteksta podataka, potrebno je konstruisati osnove sigurnosti prethodno opisane aplikacije - autentikaciju i autorizaciju kroz članstvo. Ovo podrazumjeva trajno čuvanje podataka o korisnicima, metode za upravljanje članstvom, te korištenje podataka o korisniku za ograničenje pristupa ostalim podacima.

[69] Na primjer, na Microsoft-ovoj zvaničnoj Veb lokaciji na adresi: http://msdn.microsoft.com/en-us/data/jj592674.

Pošto je pri konstrukciji konteksta podataka korišten *Code First* pristup, zgodno je iskoristiti *CodeFirst Membership Provider*[70]. Ovaj *open source* projekat predstavlja omotač za robusni *ASP.NET Membership Provider*[71], prilagođen za *Code First* kontekst podataka. Biblioteka dodaje još dva modela i njima odgovarajuće *DbSet*-ove kontekstu podataka - *Users* i *Roles*, kao i popriličan broj metoda za upravljanje članstvom (registrovanje i brisanje korisnika, dodjeljivanje i oduzimanje uloga, promjena lozinke itd.) kroz *CodeFirstMebershipProvider* i *CodeFirstRoleProvider* klase. Pošto se pri instalaciji ovog paketa obično dobija novi kontekst podataka, dovoljno je samo prekopirati *DbSet*-ove u već konstruisan kontekst, kao i *fluent API* kôd u *OnModelCreating* metodi koji uspostavlja "n na m" vezu između tabela *Users* i *Roles*:

```
// Konfiguriše vezu "n na m" između entiteta User i Role bez ručnog definisanja
// pomoćne tabele
modelBuilder.Entity<User>()
   .HasMany(u => u.Roles)
   .WithMany(r => r.Users)
       .Map(m =>
           {
               m.MapLeftKey("RoleId");
               m.MapRightKey("UserId");
               m.ToTable("RoleUsers");
           }
       );
```

Restrikcija pristupa *ASP.NET* kontrolerima se vrši pomoću *AuthorizeAttribute* i *AllowAnnonimous* anotacija. Metoda kontrolera notirana *AuthorizeAttribute* anotacijom zahtjeva prijavljenog korisnika, u protivnom vrši redirekciju na stranicu za prijavljivanje. Ukoliko se sâm kontroler notira *AuthorizeAttribute* anotacijom, anotacija se primjenjuje na sve metode kontrolera. Ukoliko se metoda kontrolera sa *AuthorizeAttribute* anotacijom notira *AllowAnnonimous* anotacijom, toj metodi može pristupiti i neprijavljen korisnik. Pored toga, *AuthorizeAttribute* može ograničiti dostupnost kontrolera (metode) samo korisnicima sa određenom ulogom, tako što mu u konstruktoru proslijede nazivi uloga:

```
[Authorize(Roles = "admin, sekretar")]
```

[70] https://codefirstmembership.codeplex.com/
[71] http://msdn.microsoft.com/en-us/library/ms731049%28v=vs.110%29.aspx

Pored toga, unutar koda kontrolera se uvijek može pristupiti trenutnom korisniku pomoću atributa *User* klase *Controller*. Može se ručno provjeriti da li je korisnik prijavljen:

```
bool isAuthenticated = User.Identity.IsAuthenticated;
```

ili pristupiti njegovom korisničkom imenu, pa pomoću *CodeFirstMebershipProvider* i *CodeFirstRoleProvider* klasa praviti proizvoljnu autentikacionu i autorizacionu logiku:

```
string username = User.Identity.Name;
string[] roles = codeFirstRoleProvider.GetRolesForUser(username);
```

Da bi *ASP.NET* koristio *CodeFirst Membership Provider*, u *Web.config* datoteci pod *system.web* tagom neophodno je navesti:

```
<system.web>
    <membership defaultProvider="CodeFirstMembershipProvider">
        <providers>
            <clear />
            <add name="CodeFirstMembershipProvider"
                 type="CodeFirstMembershipProvider"
                 connectionStringName="DataContext" />
        </providers>
    </membership>
    <roleManager enabled="true" defaultProvider="CodeFirstRoleProvider">
        <providers>
            <clear />
            <add name="CodeFirstRoleProvider"
                 type="CodeFirstRoleProvider"
                 connectionStringName="DataContext" />
        </providers>
    </roleManager>
</system.web>
```

Model sistema je konstruisan i *EF* je generisao bazu podataka pri prvom pokretanju programa. Ali, šta se sa bazom dešava po sljedećem pokretanju programa? A šta ukoliko se naprave izmjene na modelu? Da li se baza svaki put nanovo generiše? Šta se u tom slučaju desi sa podacima unesenim u bazu? O ovome se stara inicijalizator konteksta podataka. *EF* nudi tri strategije za inicijalizaciju konteksta, izražene kroz odgovarajuće klase:

- ***CreateDatabaseIfNotExists***: Podrazumjevana strategija. Pri prvom pokretanju programa se generiše baza, a u slučaju promjene modela izbacuje izuzetak.

- ***DropCreateDatabaseIfModelChanges***: Generisanu bazu briše i ponovo generiše ukoliko nastanu promjene nad modelom.
- ***DropCreateDatabaseAlways***: Pri svakom pokretanju programa baza se nanovo generiše.

Ukoliko ni jedna od ovih strategija ne odgovara, uvijek se može konstruisati novi inicijalizator. Obično se konstruiše klasa koja nasljeđuje jednu od gore navedenih klasa (ili implementira *IDatabaseInitializer<TContext>* interfejs) i predefiniše metodu *InitializeDatabase(TContext)*, u kojoj se izvršavaju potrebni *SQL* upiti.

Pored toga, zgodno je u inicijalizatoru predefinisati metodu *Seed(TContext)*. Podrazumjevana metoda ne radi ništa, a koristi se da u tek generisanu bazu unese i neke podatke. Ovo je veoma korisno u razvojnom periodu, kada se model često mijenja i baza se nanovo generiše, pa program uvijek ima neke testne podatke na raspolaganju. Takođe, ova se metoda može iskoristiti za inicijalizovanje podataka pri puštanju programa u pogon (inicijalizacija uloga, registrovanje administratora i dr.).

```
public class DataContextInitializer : DropCreateDatabaseIfModelChanges<DataContext>
{
    public void InitializeDatabase(DataContext context)
    {
        base.InitializeDatabase(context);
        context.Database.ExecuteSqlCommand(
            @"ALTER TABLE [StudyProgram]
              ADD CONSTRAINT AK_StudyProgram UNIQUE (Name);"
        );
    }
    protected override void Seed(DataContext context)
    {
        var roleProvider = new CodeFirstRoleProvider();
        roleProvider.CreateRole("admin");
        roleProvider.CreateRole("sekretar");

        var membershipProvider = new CodeFirstMembershipProvider();
        membershipProvider.CreateAccount("admin", "admin123", true);
        roleProvider.AddUsersToRoles(new[] { "admin" }, rp.GetAllRoles());

        context.StudyPrograms
            .Add(new StudyProgram { Name = "Математика и рачунарство" });
        context.SaveChanges();
    }
}
```

Inicijalizaciju je najintuitivnije zakazati u *OnModelCreating* metodi konteksta podataka, da se i konstrukcija baze i punjenje podacima zakaže na istom mjestu:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    Database.SetInitializer(new DataContextInitializer());
    ...
}
```

No, isto se može uraditi i u *Web.config* konfiguracionoj datoteci, dodajući u *appSettings* tag sljedeće:

```
<appSettings>
    <add key="DatabaseInitializerForType SekretarskiModul.Context.DataContext,
              SekretarskiModul"
         value="SekretarskiModul.Context.DataContextInitializer, SekretarskiModul" />
</appSettings>
```

Prikazana je konstrukcija funkcionalnog konteksta podataka koji komunicira sa bazom podataka. U sljedećem potpoglavlju će biti prikazana konstrukcija *API* servisa koji služe za komunikaciju klijentskih aplikacija sa konstruisanim kontekstom podataka.

5.2. *API* servisi

API servisi su dodirna tačka serverske aplikacije sa spoljašnjim svijetom (klijentskim aplikacijama). Najviše kontrole nad *HTTP* komunikacijom se u *ASP.NET* aplikaciji može ostvariti pomoću *WebAPI* kontrolera, ali i tradicionalni *MVC* kontroleri mogu dobro poslužiti u ovu svrhu. Za demonstraciju će biti prikazana konstrukcija dva kontrolera: *WebAPI* kontroler za pristup podacima i *MVC* kontroler za pristup sistemu članstva.

```
[Authorize]
public class DataController : ApiController
{
    protected readonly DataContext _context = new DataContext();
    [HttpGet]
    public User CurrentUser()
    {
        string username = User.Identity.Name;
```

```
        User user = _context.Users.FirstOrDefault(u => u.Username == username);
        if (user == null)
            throw new HttpResponseException(HttpStatusCode.NotFound);
        return user;
    }

    [HttpGet]
    [Authorize(Roles = "admin")]
    public IEnumerable<Role> Roles()
    {
        return _context.Roles;
    }

    [HttpGet]
    [AllowAnonymous]
    public Course Course(int id)
    {
        Course course = _context.Courses.FirstOrDefault(c => c.Id == id);
        if (course == null)
        {
            HttpResponseMessage response = new HttpResponseMessage()
            {
                StatusCode = HttpStatusCode.NotFound,
                ReasonPhrase = "Predmet nije pronađen"
            };
            throw new HttpResponseException(response);
        }
        return course;
    }

    [HttpDelete]
    public void Teacher(int id)
    {
        Teacher teacher = _context.Teachers.FirstOrDefault(t => t.Id == id);
        if (teacher != null)
        {
            _context.Teachers.Remove(teacher);
            _context.SaveChanges();
        }
    }
}
```

WebAPI kontroler u primjeru demonstrira neke načine pristupa kontekstu podataka. Čitav kontroler zahtjeva prijavljenog korisnika. *CurrentUser* metoda odgovara na *Get* zahtjeve i autentikovanom korisniku šalje sve podatke o njegovom korisničkom nalogu. *Users* metoda demonstrira dodatnu autorizaciju - korisnicima u ulozi “admin” šalje kolekciju podataka o svim korisnicima sistema. Metoda *Course* ne zahtjeva prijavu i vraća podatke o predmetu sa proslijeđenim *Id*-jem. Metoda *Teacher* odgovara na *Delete* zahtjeve i briše nastavnika sa proslijeđenim *Id*-jem. Takođe, kontroler u primjeru demonstrira korištenje *HTTP* statusnih

kodova za komunikaciju sa klijentom – metoda *Course* vraća odgovor sa kodom 404 i porukom ukoliko nije pronađen predmet sa proslijeđenim *Id*-jem.

```
[Authorize]
public class AccountController : Controller
{
    [HttpPost]
    [AllowAnonymous]
    public JsonResult JsonCheckLoginState()
    {
        if (User.Identity.IsAuthenticated)
        {
            var rp = new CodeFirstRoleProvider();
            var username = User.Identity.Name;
            var roles = rp.GetRolesForUser(username);
            return Json(new { loggedIn = true, username, roles });
        }
        return Json(new { loggedIn = false });
    }

    [AllowAnonymous]
    [HttpPost]
    public JsonResult JsonLogin(LoginModel model, string returnUrl)
    {
        if (ModelState.IsValid)
        {
            if (WebSecurity.Login(model.UserName, model.Password, model.RememberMe))
            {
                FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
                var rp = new CodeFirstRoleProvider();
                var username = model.UserName;
                var roles = rp.GetRolesForUser(username);
                return Json(
                    new
                    {
                        success = true,
                        redirect = returnUrl,
                        username,
                        roles
                    }
                );
            }
            ModelState.AddModelError("",
                "Korisničko ime ili lozinka koje ste proslijedili nije validna");
         }
         return Json(new { errors = GetErrorsFromModelState() });
     }

     [HttpPost]
     [Authorize(Roles = "admin")]
     public JsonResult AddRolesToUsers(MenageRolesModel model, string returnUrl)
     {
```

```
            if (ModelState.IsValid)
            {
                var rp = new CodeFirstRoleProvider();
                rp.AddUsersToRoles(model.Users, model.Roles);
                return Json(new { success = true, redirect = returnUrl });
            }
            return Json(new { errors = GetErrorsFromModelState() });
        }

        private IEnumerable<string> GetErrorsFromModelState()
        {
            return ModelState.SelectMany(
                x => x.Value.Errors.Select(error => error.ErrorMessage)
            );
        }
    }
```

Kontroler u primjeru je tradicionalni *MVC* kontroler, koji pristupa sistemu članstva i klijentu šalje odgovore u *JSON* formatu. Može se primjetiti da se koriste funkcionalnosti na koje su navikli programeri tradicionalnih *ASP.NET MVC* aplikacija: *ModelState* za validaciju zahtjeva, *WebSecurity* za pristup autentikacionom sistemu, *FormsAuthentication* za upravljanje kolačićima; jedina nova stvar je što kontroler ne generiše *HTML* od odgovarajućeg pogleda (*View*), već odgovor šalje kao *JSON* objekat.

Dakle, i *WebAPI* i tradicionalni *MVC* kontroler lijepo služe kao *API* servisi sistema. Oba imaju svoje prednosti: *MVC* kontroler robusan sistem *MC* dijela *MVC* aplikacije, *WebAPI* kontroler čistoću *HTTP* komunikacije i automatsku serijalizaciju odgovora u *JSON* formatu itd. Izbor je na programeru, a često se svodi na prethodno poznavanje tehnologije.

Klijentska aplikacija poziva *API* servise pomoću odgovarajućeg *URL*-a. *ASP.NET MVC* razvojni okvir sadrži robusan i podesiv sistem rutiranja, koji *URL*-ove usmjerava na pozive metoda kontrolera. U *ASP.NET MVC* projektu se podešavanje rutiranja obično vrši u *RegisterRoutes(RouteCollection routes)* metodi *RouteConfig* klase, koja se nalazi u *App_Start* direktorijumu:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
```

```
            defaults:
                new
                {
                    controller = "SekretarskiModul",
                    action = "Index",
                    id = UrlParameter.Optional
                }
        );
    }
}
```

Ovako podešen ruter prihvata *URL*-ove oblika *{controller}/{action}/{id}*, gdje je *id* neobavezan parametar. Ovako će *CurrentUser* metodi *DataController* klase odgovarati *URL www.sekretarskimodul.com/data/currentUser*, po *ASP.NET MVC* konvenciji. Takođe, može se podesiti podrazumjevana rutu, koja odgovara "osnovnom" *URL*-u (*www.sekretarskimodul.com*). Upravo ovde će biti smještena ona "jedna strana" koja započinje klijentsku aplikaciju. Registrovanje ruta se obično vrši po pokretanju programa, što bi u slučaju *ASP.NET MVC* aplikacije bilo u *Application_Start* metodi *Global.asax.cs* datoteke:

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        RouteConfig.RegisterRoutes(RouteTable.Routes);
    }
}
```

Rasterećenoj serverskoj strani SPA obično neće trebati komplikovanije rutiranje. Za neki komleksniji slučaj, više informacija o *ASP.NET MVC* rutiranju se može naći na Vebu[72].

Ovaj primjer predstavlja okosnicu za serverski dio SPA. Ovako organizovana serverska aplikacija je pripremljena za interakciju sa klijentima. Pomoću *API* servisa je otvorena za komunikaciju, trajno čuva podatke u bazi podataka, a svaki zahtjev prolazi kroz sigurnosne provjere. U sljedećem poglavlju će biti opisana konstrukcija klijentske aplikacije koja koristi usluge konstruisanog servera.

[72] Jedna Veb lokacija koja dobro opisuje *ASP.NET* rutiranje je http://www.asp.net/mvc/tutorials/older-versions/controllers-and-routing/asp-net-mvc-routing-overview-cs.

6. Klijent kao samostalna aplikacija

SPA klijent pruža korisniku mnogo više od korisničkog interfejsa tradicionalnih Veb aplikacija. Jednom učitana sa servera, klijentska aplikacija se u potpunosti izvršava na klijentskoj mašini, a od servera po potrebi potražuje dodatne podatke i dijelove aplikacije. SPA klijent veoma liči na desktop aplikaciju, ali ipak, SPA je Veb aplikacija: distribuira se u pregledač putem *HTTP* komunikacije, a za trajno čuvanje podataka koristi udaljeni server. SPA koncept je pokušaj da se napravi spoj ova dva svijeta, desktop i Veb aplikacija, i iskoristi najbolje iz oba.

U ovom poglavlju su predstavljene ključne tačke SPA klijentske aplikacije i primjeri rješenja u dva klijentska SPA razvojna okvira: *AngularJS*[73] i *DurandalJS*[74]. Realan SPA klijent se može konstruisati i od nule, kako je demonstrirano u [19], ali korištenje zrelog razvojnog okvira, pri konstrukciji, može značajno smanjiti kompleksnost i trajanje razvoja.

6.1. SPA klijent naspram desktop aplikacija

Baš kao i desktop aplikacija, SPA klijent prati svoje stanje, sadrži lokalne podatke, reaguje na događaje i vrši interakciju sa korisnikom. Onda kada bi desktop aplikacija podatke tražila od baze podataka, SPA klijent ih traži od servera. Zbog brzine učitavanja, SPA klijent se obično konstruiše tako da se pri učitavanju aplikacije sa servera na klijent učita upravo koliko je potrebno za inicijalno funkcionisanje aplikacije: ljuska, osnovni radni i pomoćni moduli i podaci, a ostatak se po potrebi učitava sa servera. No, ukoliko to konkretna aplikacija zahtjeva, SPA klijent se može konstruisati tako da se aplikacija čitava učita pri prvom otvaranju i ne zahtjeva dalju komunikaciju sa serverom. Ovako će aplikacija, jednom učitana, moći raditi i bez konekcije na Internet, ponašajući se kao desktop aplikacija.

Neke od prednosti SPA nad desktop aplikacijama:

- **Mogućnost pristupa sa različitih platformi:** Desktop aplikacija obično zahtjeva kompajliranje na mašini na kojoj se izvršava. SPA se u potpunosti izvršava u pregledaču, koji je dostupan na svim savremenim softverskim platformama. Prateći najbolje prakse

[73] https://angularjs.org/
[74] http://durandaljs.com/

razvoja za različite pregledače i uređaje, SPA klijent je moguće pokrenuti na skoro svim savremenim računarima.

- **Ne zahtjeva instalaciju:** Kao i za svaku Veb aplikaciju, za pristup SPA dovoljno je otvoriti odgovarajući *URL* u pregledaču.
- **Ne zahtjeva unaprijeđivanje verzije:** Pošto se svaki put ponovo distribuira klijentu, korisnik uvijek dobija najnoviju verziju aplikacije[75].
- **Distribuirana arhitektura:** Podaci se trajno čuvaju na udaljenom serveru, a ne lokalno na konkretnoj mašini. Postići isto u desktop aplikaciji često komplikuje konstrukciju, dok je za razvoj Veb aplikacija distribuirana arhitektura podrazumjevana.

Neke od prednosti desktop aplikacija nad SPA klijentima:

- **Kraći odziv:** *JavaScript* se ne izvršava brzo kao ekvivalentan kompajliran kôd, mada se razlika smanjuje stalnim naporima proizvođača pregledača. Takođe, komunikacija desktop aplikacije sa lokalnom bazom je obično značajno brža i pouzdanija nego komunikacija SPA klijenta sa serverom.
- **Rad bez konekcije na Internet:** Kao što je već rečeno, SPA klijent je po potrebi moguće konstruisati da, nakon prvog učitavanja, ne zahtjeva dalju komunikaciju sa serverom. No ipak, za samo pokretanje je neophodna konekcija na Internet. Realniji scenario je da se SPA klijent ne učita kompletan pri prvom učitavanju, te da od servera očekuje dostavljanje dodatnih podataka i dijelova aplikacije. Takođe, realniji scenario podrazumjeva trajno čuvanje podataka i prijavljivanje korisnika, za šta je Internet konekcija neophodna. Iako ovaj argument ide u prilog desktop aplikacijama, sve veća prisutnost i brzina Internet komunikacije ga čini manje bitnim.
- **Veći pristup klijentskom sistemu:** Iz sigurnosnih razloga Veb aplikacija nema pristup čitavom klijentskom sistemu. Sa druge strane, desktop aplikacije mogu veoma lako da pristupe sistemu datoteka, kameri, *GPS* uređaju, mrežnom uređaju itd. lokalnog sistema[76]. [34]

[75] Ovde se mora u obzir uzeti keširanje u pregledaču. Sasvim je moguć scenario u kome pregledač koristi keširanu zastarjelu verziju skripte ili *HTML* dokumenta, umjesto da najnoviju učita sa servera. No ovo se lako rješava praćenjem verzija datoteka, dodajući *'?version=123'* na kraj *URL*-a.
[76] Nadolazeći Veb standardi predviđaju premošćavanje i ovih nedostataka.

6.2. Ljuska

JavaScript u pregledaču nije prvobitno zamišljen za konstrukciju većih aplikacija, već za kraće skripte koje dodaju dinamičnost u inače statičan *HTML* dokument. Stoga, *JavaScript* nema ugrađen sistem definisanja prostora imena ili modula. Takođe, promjenljive nisu vidljive na nivou bloka, već na nivou funkcije u kojoj su definisane. Posljedica ovoga je da promjenljive koje nisu lokalne za neku funkciju postaju globalne, što je čest uzrok bagova. Redoslijed uključivanja skripti u dokument može da utiče da tok izvršavanja programa, što može biti pogubno za veće aplikacije. Da bi se prevazišli ovi problemi, neophodno je konstruisati sistem koji će voditi računa o životnom ciklusu SPA. Ovaj sistem se često naziva ljuskom (eng. *shell*).

Ljuska daje aplikaciji strukturu i zauzima centralno mjesto u SPA arhitekturi sa klijentske strane. Njena glavna obaveza je da prati stanje aplikacije i u zavisno od njega koordiniše funkcionisanje ostatka aplikacije [19].

Prateći najbolje prakse dizajniranja softvera, klijentsku stranu SPA treba organizovati kao skup slabo spregnutih modula, svaki sa striktno odvojenom odgovornošću. Pored opštih prednosti modularnog koda (lakše paralelno razvijanje, povećanje ponovne iskoristivosti, povećanje čitkosti, lakše testiranje itd.), dijelove SPA je mnogo lakše dostaviti pregledaču, a ljusci je mnogo lakše da njima koordiniše, ako su podijeljeni u striktno odvojene module. Zbog navedenih nedostataka *JavaScript* jezika[77] neophodno je poslužiti se programerskom finesom, koja je demonstrirana u sljedećem primjeru:

```
var myModule = (function () {
    var myPrivateVariable = 5;
    var myPublicVariable = 'Hello';
    var myPrivateFunction = function () {
       return myPrivateVariable + 7;
    };
    var myPublicFunction = function () {
       return myPrivateFunction() * 2;
    }
    return {
       myPublicVariable: myPublicVariable,
       myPublicFunction: myPublicFunction
    };
})();
```

[77] Čije otklanjanje je najavljeno u nadolazećim Veb standardima.

Skripta sa ovim kodom, učitana u *HTML* dokument, registruje globalnu promjenljivu *myModule*, koja eksportuje jedan objekat. Semantička konstrukcija koje je ovde iskorištena je tzv. „samoizvršavajuća funkcija“ ili „anonimno zatvorenje“. Naime, definisana je funkcija, pa potom odmah izvršena. Sve varijable i funkcije definisane u njoj su lokalne i ne „prljaju“ globalni prostor imena. Sa druge strane, ukoliko varijabla ili funkcija treba biti vidljiva u ostatku programa, može biti eksportovana u objektu kojeg samoizvršavajuća funkcija vraća, kao što je u primjeru postupljeno sa *myPublicVariable* i *myPublicFunction*. Njima se može pristupiti preko *myModule* globalne varijable:

```
myModule.myPublicFunction();
```

Korištenjem samoizvršavajuće funkcije, lako se vodi računa i o međusobnoj zavisnosti modula:

```
var myOtherModule = (function (dependencyModule, globalNamespace, $) {
    var localCopy = dependencyModule.myPublicVariable;
    globalNamespace.globalVariable = 5;
    return {
        showAllDivs: function () {
            $("div").show();
        }
    };
})(myModule, window, jQuery);
```

Kao što se u primjeru vidi, samoizvršavajućoj funkciji se kao agrumenti mogu proslijediti drugi moduli, globalni objekti (kao *window* objekat) i objekti iz *third-party* biblioteka. U definiciji funkcije se ovim objektima mogu dati drugačija imena, da ne bi došlo do zabune pri pisanju koda. Kao što se iz primjera vidi, u modulu se mogu koristiti varijable i funkcije koje eksportuju drugi moduli, od kojih trenutni zavisi. Takođe, demonstriran je način na koji modul može registrovati globalnu varijablu, korištenjem *window* objekta, mada to nije preporučljivo. Najbolja praksa preporučuje organizovanje koda u manje module, koji eksportuju samo jedan javni interfejs. Ovako imena varijabli ostaju lokalna, detalji implementacije su skriveni van modula, a zavisnosti su eksplicitno navedene, pa su prepravljanje kôda i testiranje mnogo lakši [29].

CommonJS[78] je projekat pokrenut 2009. (prvobitno pod nazivom *ServerJS*), sa ciljem uspostavljanja ekosistema za *JavaScript* razvoj, ne samo u pregledačima, već u serverskim, desktop i aplikacijama za komandnu liniju. Krajnji proizvod ovog otvorenog projekta su specifikacije za rješavanje konkretnih problema u programiranju *JavaScript* aplikacija. Među njima je i specifikacija za module[79], implementirana npr. u *Node.js* serverskoj platformi. No ona je razvijena imajući u vidu više serversko okruženje i nije bila prikladna za pregledač. Stoga je razvijen *Modules/Transport* niz specifikacija, koje su rješavale problem asinhronog učitavanja modula i razrješavanja zavisnosti. *Modules/Transport/C* specifikacija, koja je kasnije izrasla u samostalnu *Modules/AsynchronousDefinition* specifikaciju ili *AMD API* (eng. *Asynchronous Module Definition*), je implementirana u *RequireJS* biblioteci[80]. „Ispod haube", *RequireJS* module obavija samoizvršavajućom funkcijom, baš kao u prethodnim primjerima.

AngularJS, SPA razvojni okvir razvijan od strane *Google*-a, sadrži svoj sistem modula, dok Durandal, SPA razvojni okvir kojeg razvija kompanija *Blue Spire*[81], za modularizaciju kôda koristi *RequireJS*. Oba sistema obezbjeđuju modularizaciju koda i ubrizgavanje zavisnosti (eng. *Dependency Injection*, u daljem tekstu *DI*), no *RequireJS* ima prednost potpunog asinhronog učitavanja modula i razrješavanja zavisnosti. Naime, kada se na Veb stranici koristi *RequireJS*, jedino se ona (i ostale *third-party* biblioteke) učita sinhrono sa stranicom. Ostatak korisničkog koda se učitava asinhrono i to po potrebi – kada je modul neposredno zahtjevan ili neki drugi modul zavisi od njega. Takođe, jednom učitan modul ne zahtjeva ponovno učitavanje sa servera. S obzirom da je *RequireJS* samostalna biblioteka, sasvim se lijepo može koristiti i sa *AngularJS*-om, mada to u ovom radu neće biti obrađivano.

U sljedećim primjerima će biti demonstrirana konstrukcija modūlā u navedena dva razvojna okvira, i to upravo pri konstrukciji ljuske kao glavnog modula.

[78] http://www.commonjs.org/
[79] http://wiki.commonjs.org/wiki/Modules
[80] http://requirejs.org/
[81] http://bluespire.com/

6.2.1. Osnovni *AngularJS* primjer

U *AngularJS* primjeru se pretpostavlja da je preuzeta *angular.js* datoteka, koja sadrži *AngularJS* biblioteku i da je smještena u direktorijum *lib*, *main.js* skriptu u *app* driektorijumu, kao i *index.html* dokument u korjenom direktorijumu.

index.html

```
<!DOCTYPE html>
<html ng-app="mySPA">
<head>
    <meta charset="utf-8">
    <title>Sekretarski Modul - Angular</title>
</head>
<body ng-controller="homeController as homeCtrl">
    Vaše ime:
    <input type="text" ng-model="homeCtrl.name" />
    Zdravo {{ homeCtrl.name }}
    <button ng-click="homeCtrl.greet()">Pozdrav</button>
    <script src="lib/angular.js"></script>
    <script src="app/main.js"></script>
</body>
</html>
```

app/main.js

```
var app = angular.module('mySPA', []);
app.controller('homeController', function () {
    this.name = '';
    this.greet = function () {
        alert('Dobrodošli u Sekretarski Modul ' + this.name);
    };
});
```

AngularJS intenzivno koristi fabrički šablon[82]. U *main.js* datoteci je konstruisan jedan modul, pozivom fabričke metode *angular.module*. Njoj se proslijeđuje identifikacioni naziv modula i niz naziva modūlā od kojih novi modul zavisi. Modul u primjeru nema zavisnosti, pa je proslijeđen prazan niz.

[82] Šablon u dizajnu softvera koji se odnosi na pristup kreiranju objekata. Po njemu se objekti ne instanciraju direktno, već se poziva metoda koja instancira objekat. Ovako programer ne mora da zna tačan tip objekta koji se instancira, jer sam ne poziva konstruktor.

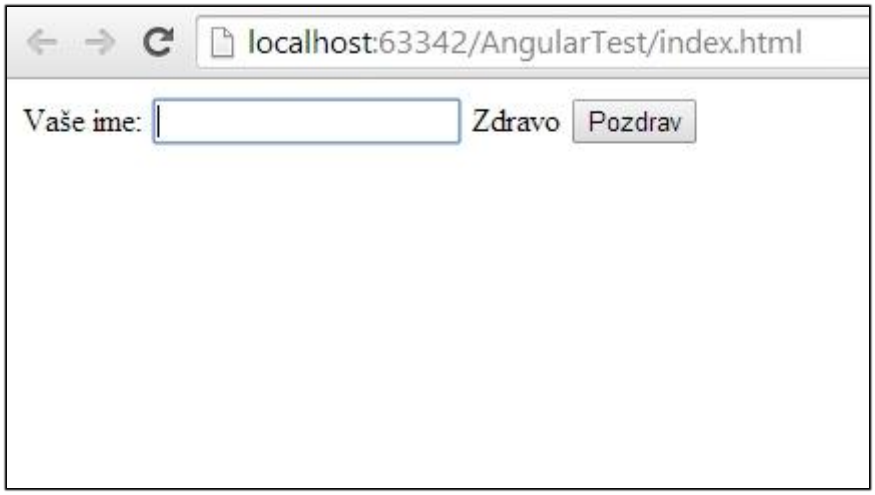
AngularJS bibiloteka definiše tzv. direktive - *HTML* oznake (tagove, atribute, klase itd.) koji dozvoljavaju *AngularJS* biblioteci da vrši interakciju sa *HTML* dokumentom. Korištenjem direktiva *HTML* dokument ne izražava samo strukturu korisničkog interfejsa, već i ponašanje aplikacije. *AngularJS* dolazi sa dosta ugrađenih direktiva, čiji naziv mahom počinje sa *ng*, ali nudi i jednostavan način konstrukcije sopstvenih direktiva. Tako se, na primjer, opseg dokumenta na koji *AngularJS* ima uticaja označava sa *ng-app* atributom, kome se proslijedi naziv glavnog modula programa, u ovom slučaju *'mySPA'*.

AngularJS primjenjuje *MVC* paradigmu. Poglede (eng. *View*) predstavljaju *HTML* šabloni (dokumenti označeni direktivama), kontroleri (eng. *Controller*) su funkcionalnosti koje sadrže poslovnu logiku i izlažu pogledu podatke (eng. *Model*) i funkcije za njihovu manipulaciju. Kontroler se vezuje za konkretan modul i definiše se pozivom njegove fabričke metode *controller*. Njoj se proslijeđuje naziv kontrolera (*'homeController'* u ovom slučaju) i definiciona funkcija. Svi podaci i funkcije koje je potrebno izložiti pogledu treba zakačiti na *this* referencu. „Ispod haube“, *AngularJS* definicionu funkciju obavija samoizvršavajućom funkcijom, slično kao u prikazanim primjerima. Sve podatke koje kontroler izlaže, *AngularJS* obavija sistemom za detekciju promjena, primjenom *observer* šablona[83]. Ovim je omogućeno dvosmjerno povezivanje podataka (eng. *data binding*) između modela i *HTML* dokumenta.

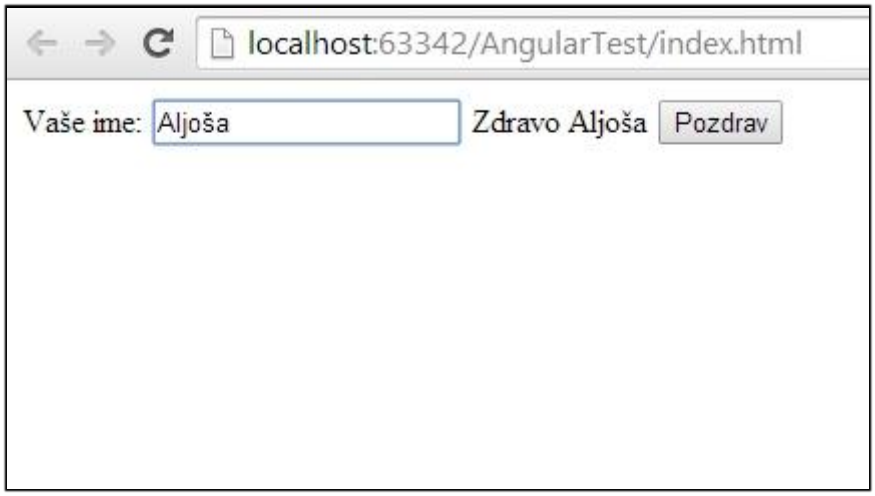
Povezivanje podataka je demonstrirano u primjeru. Pomoću *ng-controller* direktive je naznačeno da će *homeController* uticati na čitav *body* tag dokumenta, a dat mu je *homeCtrl* pseudonim za lakše manipulisanje. Kontroler pogledu izlaže promjenljivu *name* i funkciju *greet*. Pogled (*HTML* šablon) definiše *input* polje, koje kao model koristi *name* varijablu (ima naznačen atribut *ng-model="homeCtrl.name"*). Dalje u dokumentu se u dvostrukim vitičastim zagradama nalazi *homeCtrl.name*. Ovom sintaksom se u *AngularJS*-u definišu izrazi unutar *HTML* dokumenata. To su *JavaScript* izrazi koji se izračunavaju u odnosu na trenutni opseg varijabli. U ovom slučaju, ispisaće se trenutna vrijednost *name* varijable. Ukoliko korisnik u *input* polje upiše neki tekst, isti će se ispisati pored „Zdravo“. Takođe, ukoliko korisnik pritisne dugme za pozdrav, prikazaće se *pop-up* prozorčić u kojem će pisati „Dobrodošli u Sekretarski modul“, praćeno unesenim tekstom, tj. pozvaće se funkcija *greet*, jer dugme ima naznačen atribut *ng-click="homeCtrl.greet()"*. Automatsko dvosmjerno povezivanje podataka svaku promjenu

[83] Šablon u dizajnu softvera u kome objekat prati listu observera – objekata koji od njega zavise – i automatski ih obavještava o svakoj promjeni svoje vrijednosti.

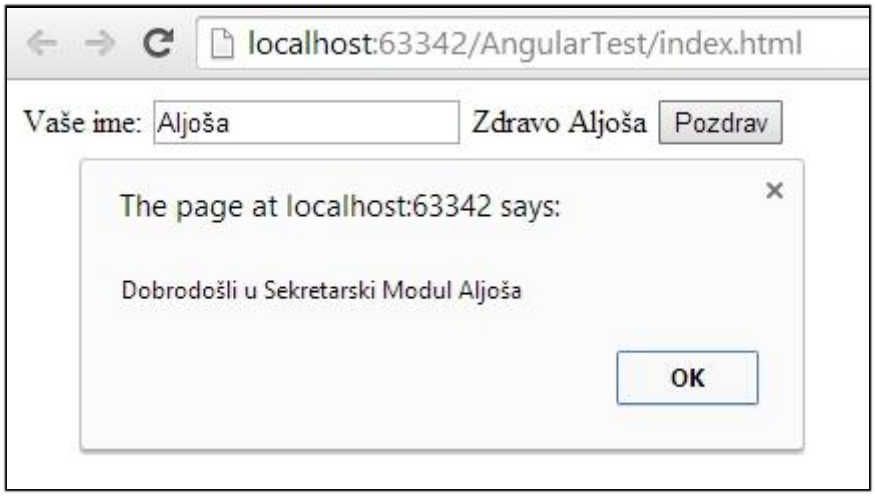
vrijednosti *input* polja čuva u varijablu *name* unutar kontrolera, a ta promjena se propagira svugdje gdje se *name* koristi; u ovom slučaju u *{{ homeCtrl.name }}* izrazu u *HTLM* dokumentu i *greet* funkciji. Svaka promjena nad *HTML* dokumentom se dešava isključivo na klijentskoj strani i nema ponovnog učitavanja dokumenta sa servera.



Slika 1 – Osnovni AngularJS primjer, pokrenut u Google Chrome pregledaču



Slika 2 – AngularJS-ovo automatsko dvosmjerno povezivanje podataka



Slika 3 – Nakon klika na dugme "Pozdrav" u AngularJS-u

6.2.2. Osnovni *Durandal* primjer

Uspostavljanje *Durandal* aplikacije zahtjeva nešto više truda nego *AngularJS* aplikacija. Dok *AngularJS* pruža mogućnosti modularne organizacije koda, kod njega se ne insistira na tome. Ovo je veoma pogodno za manje aplikacije, koji se mogu veoma brzo razviti. *Durandal* isključivo insistira na modularizaciji koda i podrazumjeva kompleksniju strukturu, uobičajenu za veće aplikacije.

U sljedećem primjeru se podrazumjeva sljedeća struktura datoteka projekta:

- Korjeni direktorijum projekta sadrži *index.html* dokument;
- Poddirektorijum *lib* korjenog direktorijuma sadrži neophodne biblioteke, svaku u zasebnom direktorijumu: *durandal*, *jquery*, *knockout* i *require*;
- Poddirektorijum *app* korjenog direktorijuma sadrži *HTML* fragmente i *JavaScript* skripte koje korisnik definiše.

index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Sekretarski Modul - Durandal</title>
</head>
<body>
    <div id="applicationHost">
    </div>
    <script src="lib/require/require.js" data-main="app/main"></script>
</body>
</html>
```

Veoma oskudni *index.html* definiše dio dokumenta gdje će korisnički interfejs aplikacije biti generisan (*div* sa *id*-jem *applicationHost*, po konvenciji), i *script* tag za učitavanje *RequireJS* biblioteke, kojoj se proslijedi ulazna tačka u aplikaciju u *data-main* atributu, *app/main.js* u ovom slučaju. Pri učitavanju skripti sa *RequireJS*-om, nije neophodno navoditi *.js* ekstenziju.

app/main.js

```
requirejs.config({
    paths: {
        'text': '../lib/require/text',
        'durandal': '../lib/durandal/js',
        'knockout': '../lib/knockout/knockout-3.1.0',
        'jquery': '../lib/jquery/jquery-1.9.1'
    }
});
define(['../lib/durandal/js/app', 'durandal/viewLocator'], function (app,
viewLocator) {
    app.start().then(function () {
        viewLocator.useConvention();
        app.setRoot('shell');
    });
});
```

Ulazna tačka aplikacije, *app/main.js*, sadrži konfiguraciju za *RequireJS* i jedan modul koji započinje aplikaciju. *RequireJS* nudi mnoge mogućnosti konfiguracije, ali u ovom slučaju su neohpodne samo osnovne. Konfiguracije se proslijeđuju funkciji *requirejs.config* u objektu. *RequireJS* datoteke traži u korjenom direktorijumu projekta. Da se ne bi svaki put morao navoditi puni put do datoteka, zgodno je definisati kratice za putanje (direktorijume i datoteke) koje se često koriste u projektu. Ovo se postiže tako što se u konfiguracionom objektu naznači

ključ *paths* koji za vrijednost ima objekat kome su ključevi nazivi putanja, a vrijednosti same putanje, relativne u odnosu na datoteku u kojoj se definišu (*app/main.js*). Ukoliko se navodi put do datoteke, nije potrebno navesti ekstenziju. *Durandal* koristi putanje navedene u primjeru, pa ih je neophodno navesti da bi aplikacija funkcionisala:

- *text* je put do *text.js RequireJS* dodatka, koji omogućuje učitavanje običnih tekstualnih datoteka, a ne samo skripti. Neophodan je da bi *Durandal* mogao asinhrono učitavati *HTML* dokumente i *CSS* stilove.
- *durandal* je put do direktorijuma *Durandal* biblioteke.
- *knockout* je put do *KnockoutJS* biblioteke koju *Durandal* koristi za generisanje *HTML*-a pomoću povezivanja podataka.
- *jquery* je put do popularne *jQuery* biblioteke, koju *Durandal* interno koristi u razne svrhe.

RequireJS registruje globalnu funkciju *define* koja služi za definsanje modula. Funkciji se proslijeđuje opcioni niz naziva modūlā od kojih novi zavisi, te definiciona funkcija. *RequireJS* generiše naziv modulu po putu do datoteke u kojoj je definisan, počevši od korjenog direktorijuma projekta. S obzirom na to, preporučeno je definisati samo jedan modul po datoteci. Gore definisan modul će dobiti naziv *app/main* (bez *.js* ekstenzije). Ukoliko modul zavisi od drugih, neophodno je navesti njihove nazive. Oni će se definicionoj funkciji proslijediti kao argumenti, i to u istom redoslijedu kako su navedeni u nizu zavisnosti. Put do potrebnog modula *RequireJS* razrješuje u odnosu na direktorijum u kojem je trenutno definisani modul, a mogu se koristiti i definisane skraćenice. Modul u primjeru zavisi od *app* i *viewLocator* modula *Durandal* biblioteke. Prvi je referisan pomoću relativnog puta, a drugi koristeći skraćenicu.

Sa stranicom će se učitati *RequireJS* bibilioteka, koja će prvo učitati *app/main.js* datoteku, jer je naznačena kao ulazna tačka programa. Po učitavanju *main* modula, *RequireJS* će razrješiti *app* i *viewLocator* zavisnosti i učitati odgovarajuće datoteke. Datoteke će biti potraživane od servera dok god sve zavisnosti ne budi razrješene. Nakon što je modul jednom učitan na klijenta, ne zahtjeva ponovno učitavanje i njegovo razrješavanje je trenutačno.

Glavni modul *Durandal* aplikacije obično služi za globalne postavke i pokretanje aplikacije. Ovako mali primjer zahtjeva samo osnovne postavke. Najprije je aplikacija startovana, a nakon toga rečeno da se koristi konvencija pri nalaženju pogleda (eng. *View*) i za korjeni modul postavljen *shell*. *Durandal* za povezivanje podataka koristi *KnockoutJS*, *MVVM*

(*Model View ViewModel*) *JavaScript* biblioteku. *MVVM* paradigma podrazumjeva „1 na 1“ onos između pogleda (*View*) i kôda odgovornog za njega (*ViewModel*, u daljem tekstu *VM*). U ovom slučaju, pogledi će biti *HTML* fragmenti, ispunjeni posebnim oznakama za povezivanje podataka, a *VM JavaScript* moduli, koji izlažu jedan javni objekat sa podacima (*Model*). S obzirom za „1 na 1“ odnos, pomenuta konvencija nalaže da se pogled i *VM* ili nalaze u istom direktorijumu ili da se pogled nalazi u direktorijumu *views*, a odgovarajući kôd u direktorijumu *viewmodels*, a da su oni u istom direktorijumu. U oba slučaja, konvencija zahtjeva da odgovarajući pogled i kôd nose isti naziv. U primjeru, to su *shell.html* i *shell.js*, u direktorijumu *app*. Ukoliko ne odgovara, ova konvencija se može prekonfigurisati.

app/shell.html

```
<div>
    Vaše ime:
    <input type="text" data-bind="value: name" />
    Zdravo <span data-bind="text: name"></span>
    <button data-bind="click: greet">Pozdrav</button>
</div>
```

app/shell.js

```
define(['require'], function (require) {
    var ko = require('knockout');
    var name = ko.observable('');

    var greet = function () {
        alert('Dobrodošli u Sekretarski Modul ' + name());
    };

    return {
        name: name,
        greet: greet
    };
});
```

Pogled u primjeru veoma liči na *AngularJS* primjer. Sadrži input polje, span tag i dugme, samo što *KnockoutJS* za povezivanje podataka koristi atribut *data-bind* na tagovima. Pomoću *value* povezivanja se vrijednost input polja vezuje za *name* varilablu koju izlaže *VM*. Takođe, unutrašnji tekst span taga će biti jednak *name* varijabli, a po kliku na dugme će se pokrenuti *greet* funkcija. *VM* je definisan kao *RequireJS* modul. U njemu je demonstriran još jedan način

navođenja zavisnosti modula. U ovom slučaju, modul zavisi od *KnockoutJS* biblioteke, a zavisnost se eksplicitno navodi u kodu, pozivanjem *require('knockout')*. Ovime *RequireJS* razrješuje traženi modul (naveden pomoću kratice) i smješta ga u varijablu *ko*, slično kako bi uradio sa nizom naziva modula i argumentima definicione funkcije. Eksplicitno navođenje zavisnosti pogoduje u slučaju da modul zavisi od dosta drugih modula, pa vođenje brige o redoslijedu naziva i argumenata postane problematično:

```
define(['require', '../main', 'services/datacontext', 'services/utilities',
        'jquery', 'jquery-ui', 'widgets/dragAndDrop', 'animation/bounce'],
    function (require, main, datacontext, utilities,
              $, jQueryUI, dragAndDrop, bounce) {

        ...

    }
);
```

se čitkije može napisati kao:

```
define(['require'],
    function (require) {

        var main = require('../main');
        var datacontext = require('services/datacontext');
        var utilities = require('services/utilities');
        var $ = require('jquery');
        var jqueryUI = require('jquery-ui');
        var dragAndDrop = require('widgets/dragAndDrop');
        var bounce = require('animation/bounce');

        ...

    }
);
```

Također, eksplicitnim navođenjem zavisnosti se modul može učitati tek onda kada je neophodan. Standardnim navođenem, zavisnosti se razrješuju pri učitavanju modula. Eksplicitnim navođenjem, zavisnost se razrješuje tek po izvršavanju *require* funkcije. To omogućuje da se neesencijalan modul zatraži tek onda kada je baš neophodan:

```
define(['require'],
    function (require) {

        ...

        var fn = function () {
            if (additionalModuleRequired) {
                var additionalModule = require('additionalModule');

                ...

            }
        };
    }
);
```

Definisani modul izlaže promjenljivu *name* i funkciju *greet*. Pomoću funkcije *observable* bibilioteke *KnockoutJS* se promjenljiva *name* označava kao „nadgledana“ (eng. observable), što je smješta u sistem za praćenje promjena, slično kao sa elementima *AngularJS* kontrolera. Nadgledana promjenljiva može da se inicijalizuje nekom vrijednošću, kao u primjeru sa *name* varijablom, tako što se funkciji *observable* proslijedi vrijednost.

Rezultujuća aplikacija će izgledati veoma slično *AngularJS* primjeru. Jedina veća razlika je što se vrijednost *name* nadgledane varijable neće promjeniti pri svakoj promjeni vrijednosti input polja, već tek kad polje izgubi fokus. Ovo je podrazumjevano ponašanje *KnockoutJS*-a, a funkcionalnost kao u *AngularJS* primjeru se može postići navođenjem *valueUpdate* opcije u *data-bind* atributu:

```
<input type="text" data-bind="value: name, valueUpdate: 'afterkeydown'" />
```

Izgled i funkcionalnost ovog jednostavnog *Durandal* primjera će biti identični onom iz *AngularJS* primjera. Oba primjera ilustruju osnovne karakteristike SPA. Na klijentu se učitava jedan dokument, *index.html*, koji započinje aplikaciju. Nakon toga, svo generisanje *HTML*-a i poslovna logika su na klijentu, a dodatne *HTML* i *JavaScript* datoteke se asinhrono učitavaju sa servera, po potrebi. U oba slučaja, većinu obaveza ljuske obavlja razvojni okvir, a programeru se dozvoljava konfigursanje aplikacije prema konkretnoj potrebi.

6.3. Upravljanje stanjem aplikacije

U računarstvu, pod stanjem se podrazumjeva jedinstvena konfiguracija podataka u aplikaciji [19]. Većina desktop aplikacija održava stanje na neki način. Po potrebi korisniku pružaju mogućnosti čuvanja i učitavanja stanja radi njegove postojanosti između sesija korištenja aplikacije. Takođe, neke desktop aplikacije nude *undo* i *redo* mogućnosti, koje dozvoljavaju lak prolazak kroz stanja. Sa druge strane, stanje Veb aplikacije se obično smješta u *URL*, a svi moderni pregledači nude *back*, *forward* i *bookmark* mogućnosti, kao i istoriju pregledanja. Pri razvijanju desktop aplikacije, programer može napraviti izbor da ne implementira čuvanje i prolazak kroz stanja, ali s obzirom da su korisnici pregledača odavno navikli na *back*, *forward*, *bookmark* i *history* opcije, SPA programer nema takav izbor. Korisnici od ovih opcija očekuju sljedeće:

- *Back* – povratak na prošlo stanje;
- *Forward* – obnova stanja prije posljednje *back* operacije;
- *Bookmark* – spašavanje trenutnog stanja za kasnije korištenje;
- *History* – pregled stānjā kroz koja je prošla aplikacija.

Nekonzistentna integracija ovih opcija u SPA može predstavljati izuzetno loše iskustvo za korisnika.

Kao i za ostale Veb aplikacije, i za SPA je *URL* pogodno mjesto za bilježenje stanja. Validan *URL* se sastoji iz sljedećih djelova:
<scheme>://<domain>[:<port>]/<path>[?<query>][#<fragment>]

- *scheme* – protokol korišten u komunikaciji, definiše način konekcije (npr. *http*, *https*, *ftp*...);
- *domain* – naziv ili *IP* adresa ciljnog servera, definiše ciljno mjesto konekcije (npr. *www.google.com*, *173.194.34.5*);
- *port* – opcioni broj *TCP/IP* porta koji se koristi za kominukaciju;
- *path* – putanja do traženog resursa, niz segmenata odvojenih znakom '/';
- *query* – opcioni dio koji se proslijeđuje serverskoj aplikaciji kao ulazni podatak. Iako nije obavezno, obično se sastoji od parova ključa i vrijednosti, odvojenih znakom '&' (npr. *?first_name=Marko&last_name=Marković*);

- *fragment* – opcioni dio koji referiše na poseban dio traženog resursa.

Znaci dozvoljeni u *URL*-u su velika i mala slova latiničnog alfabeta, brojevi, te sljedeći specijalni znaci: - _ . ~ ! * ' ( ) ; : @ & = + $, / ? % # [ ] [35].

Za razliku od ostatka *URL*-a, fragment je isključivo klijentska oznaka. Kada klijent potražuje resurs pomoću *URL*-a sa fragmentskim djelom, server generiše i proslijeđuje dokument, a pretraživač na klijentu se stara o pronalasku dijela dokumenta na koji fragment referiše. Takođe, promjena fragmenta ne pokreće ponovno učitavanje stranice, ali se zato bilježi u istoriju pregledanja. Sve nakon prvog znaka *'#'* u *URL*-u se smatra za fragment, on nema striktno definisanu sintaksu. S obzirom na veliki broj ponuđenih znakova, odgovarajućim formatiranjem se u fragment može smjestiti poprilično komplikovana struktura. Sve ovo ga čini veoma pogodnim za bilježenje stanja SPA.

Fragmentski dio *URL*-a se obično koristi da olakša navigaciju kroz veoma dugačak dokument. Značajnim dijelovima dokumenta (npr. naslovima) se dodjele identifikatori, pa pri promjeni vrijednost fragmentskog dijela pregledač prikaže odgovarajući dio dokumenta. Ovo ponašanje pregledača se može poništiti presretanjem promjene vrijednosti fragmenta kroz *JavaScript* kôd.

I *AngularJS* i *Durandal* biblioteke koriste ovu tehniku za bilježenje stanja aplikacije u *URL*. Imitirajući klasične serverske Veb aplikacije, fragmentski dio *URL*-a se formatira u rutu, slično *path* dijelu *URL*-a, sa opcionim *query* dijelom, npr.:

http://sekretarski-modul.apphb.com/#/schedules/view?studyProgramId=1

No, s obzirom na veliki broj mogućih manjih stanja u kojima SPA može da se nađe, pri izboru onih koja će biti zapisana u *URL* programer treba da razmotri sljedeće:

- Koliko je vjerovatno da će korisnik željeti stanje sačuvati kao *bookmark*?
- Koliko je vjerovatno da će korisnik željeti da stanje aplikacije vrati na prošlo pomoću *back* opcije?
- Koliko je operacija povratka stanja aplikacije skupa, ukoliko je uopšte moguća?

Posljednja stavka je posebno bitna u slučajevima kada aplikacija vrši interakciju sa sistemima treće strane. Na primjer, veoma je teško otkazati obavljenu *online* kupovinu za koju je korišten *PayPal* za novčanu transakciju, klikom na *Back* dugme [19].

6.4. Klijentska aplikacija sa stanjem

U ovom poglavlju biće prikazani primjeri potpune, iako male, *AngularJS* i *Durandal* klijentske aplikacije. Aplikacija prati svoje stanje i komunicira sa serverskom aplikacijom pomoću *AJAX* poziva. Sastojaće se od glavnog modula, tri radna modula i pomoćnog servisa. Prvi radni modul predstavlja početnu stranu aplikacije i prikazivaće pozdrav i vezu ka drugom radnom modulu, koji će prikazivati listu predmeta. Klikom na element liste, aplikacija će korisnika voditi u treći radni modul, koji će prikazivati više informacija o izabranom predmetu. Dobavljanje podataka o predmetima sa servera će se vršiti kroz pomoćni servis.

6.4.1. Kompleksniji *AngularJS* primjer

Views/SekretarskiModul/Index.cshtml

```
<!DOCTYPE html>
<html ng-app="sekretarskiModul">
<head>
    <meta charset="utf-8">
    <title>Sekretarski Modul - Angular</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
                                 bootstrap/3.2.0/css/bootstrap.min.css">
    <style>
        body { max-width: 600px; }
    </style>
</head>
<body>
    <div ng-view></div>
    <script src="lib/angular.js"></script>
    <script src="lib/angular-route.js"></script>
    <script src="app/main.js"></script>
</body>
</html>
```

U ovom primjeru je u stranicu uključen popularni *Twitter Bootstrap*[84] razvojni okvir za stilizaciju Veb stranica, kao i jedan red *CSS* koda, radi ljepšeg izgleda primjera. Takođe, pored *AngularJS* bibilioteke, u projekat je uključen *angular-route* modul, koji se stara o rutiranju.

[84] http://getbootstrap.com/

U *div* tagu označenim *ng-view* direktivom će se dinamički prikazivati trenutno aktivan pogled, u zavisnosti od trenutno aktivne rute u *URL*-u. Konfiguracija ruta se vrši u glavnom modulu, najčešće u *config* bloku u kojem se obično vrše konfiguracije *AngularJS* modula:

app/main.js

```
var app = angular.module('sekretarskiModul', ['ngRoute']);

app.config(['$routeProvider', function ($routeProvider) {
    $routeProvider
        .when('/home', {
            templateUrl: 'app/views/home.html',
            controller: 'HomeController',
            controllerAs: 'homeCtrl'
        })
        .when('/courses', {
            templateUrl: 'app/views/courses.html',
            controller: 'CoursesController',
            controllerAs: 'coursesCtrl'
        })
        .when('/course/:id', {
            templateUrl: 'app/views/course.html',
            controller: 'CourseController',
            controllerAs: 'courseCtrl'
        })
        .otherwise({
            redirectTo: '/home'
        });
}]);
```

Prvo je neophodno navesti da glavni modul zavisi od *ngRoute* modula, koji je definisan u *lib/angular-route.js* biblioteci i pruža servise za upravljanje rutama. Jedan od njih je *$routeProvider* servis, koji služi za konfigurisanje ruta. Pomoću njega, u *config* bloku su registrovane tri rute sa sljedećim osobinama:

- Prva ruta će odgovarati *#/home* fragmentskom dijelu *URL*-a, pogled (*HTML* šema) koji definiše korisnički interfejs tog djela aplikacije će se nalaziti u *app/views/home.html* datoteci, a kao kontroler će se koristiti *HomeController*, koji će u pogledu biti korišten pod pseudonimom *homeCtrl*;
- Druga ruta, ekvivalentno prvoj, će odgovarati *#/courses* fragmentu, sa *app/views/courses.html* pogledom i *CoursesController* kontrolerom, sa *coursesCtrl* pseudonimom;

- Treća ruta će sadržati parametar *id*, pa će odgovarati npr. *#/course/10*, *#/course/53*, kao i *#/course/abc* fragmenu, ali ne i *#/course/12/abc* fragmenu, tj. parametar *id* će odgovarati bilo kojem nizu (*URL* validnih) karaktera, do prvog '/' karaktera. Također, za pogled će se koristiti *app/views/course.html*, a za kontroler *CourseController*, sa *courseCtrl* skraćenicom.
- Pomoću *otherwise* poziva definišemo ponašanje aplikacije u slučaju da se u *URL*-u pojavi neregistrovana ruta. U ovom slučaju će aplikacija automatski redirektovati korisnika na *#/home* rutu. Ovo je korisno zbog konzistentnog iskustva za korisnika, jer se moguće greške u unesenom *URL*-u, kao i moguće greške u vezama, ublažavaju redirektovanjem na validnu rutu.

app/views/home.html

```
<div class="jumbotron text-center">
    <h1>{{ homeCtrl.greeting }}</h1>
    <a class="btn btn-lg btn-primary" href="#/courses">Predmeti</a>
</div>
```

app/main.js

```
...

app.controller('HomeController', function () {
    this.greeting = 'Dobrodošli!';
});
```

Pošto su definisani pogled i kontroler za */home* rutu, nakon pokretanja programa u pregledaču će biti prikazano sljedeće:

Slika 4 – Početni pogled kompleksnije AngularJS aplikacije

Može se primjetiti da, iako korisnik unese samo osnovni *URL* (*localhost:8080/AngularTest)*, aplikacija će ga automatski redirektovati na *localhost:8080/AngularTest/#/home*.

S obzirom da je neophodan za dobavljanje podataka o predmetima za preostale dvije rute, sljedeće će biti prikazana konstrukcija pomoćnog servisa, koji bi u ozbiljnoj SPA predstavljao dio konteksta podataka na klijentu.

app/main.js

```
...
app.service('CourseService', ['$http', function ($http) {
    var CourseService = {
        getAll: function () {
            return $http({
                method: 'GET',
                url: Data/Courses'
            });
        },
        getById: function (id) {
            return $http({
                method: 'GET',
                url: 'Data/Course/' + id
            });
        }
    };

    return CourseService;
}]);
```

Pomoću fabričke metode *service* u modulu se registruje pomoćni servis, koji se drugim komponentama modula (kontrolerima, direktivama, drugim servisima itd.) može proslijediti kao zavisnost. Funkciji *service* se proslijeđuje naziv novog servisa i definiciona funkcija. Kao i pri definiciji kontrolera, ukoliko servis ne sadrži zavisnosti, dovoljno je proslijediti samo definicionu funkciju. Ukoliko, međutim, servis zavisi od drugih komponenti modula, neophodno je proslijediti definicioni niz, koji sadrži imena zavisnosti i kao zadnji element definicionu funkciju. Zavisnosti će biti redom ubačene kao argumenti definicione funkcije. Servis u primjeru zavisi od *AngularJS*-ovog *$http* servisa, koji vrši *AJAX* pozive ka serveru. Servis se uvijek ubacuje kao *singleton*[85] pri razrješavanju zavisnosti. Servis javno izlaže samo jedan objekat, u ovom slučaju sa dvije funkcije:

- *getAll()* – koja pokušava da sa servera učita nazive i *id*-jeve svih predmeta;
- *getById(id)* – koja pokušava sa servera da učita sve podatke o predmetu sa proslijeđenim *id*-jem.

Sada je sve spremno za konstrukciju preostala dva radna modula.

app/views/courses.html

```
<div class="text-center">
    <div class="list-group">
        <a ng-repeat="course in coursesCtrl.courses"
           href="#/course/{{ course.Id }}"
           class="list-group-item">
            {{ course.Name }}
        </a>
    </div>
</div>
```

app/main.js

```
...
app.controller('CoursesController', ['CourseService', function (CourseService) {

    var self = this;
    self.courses = [];

    CourseService.getAll()
        .success(function (data) {
            self.courses = data;
        })
```

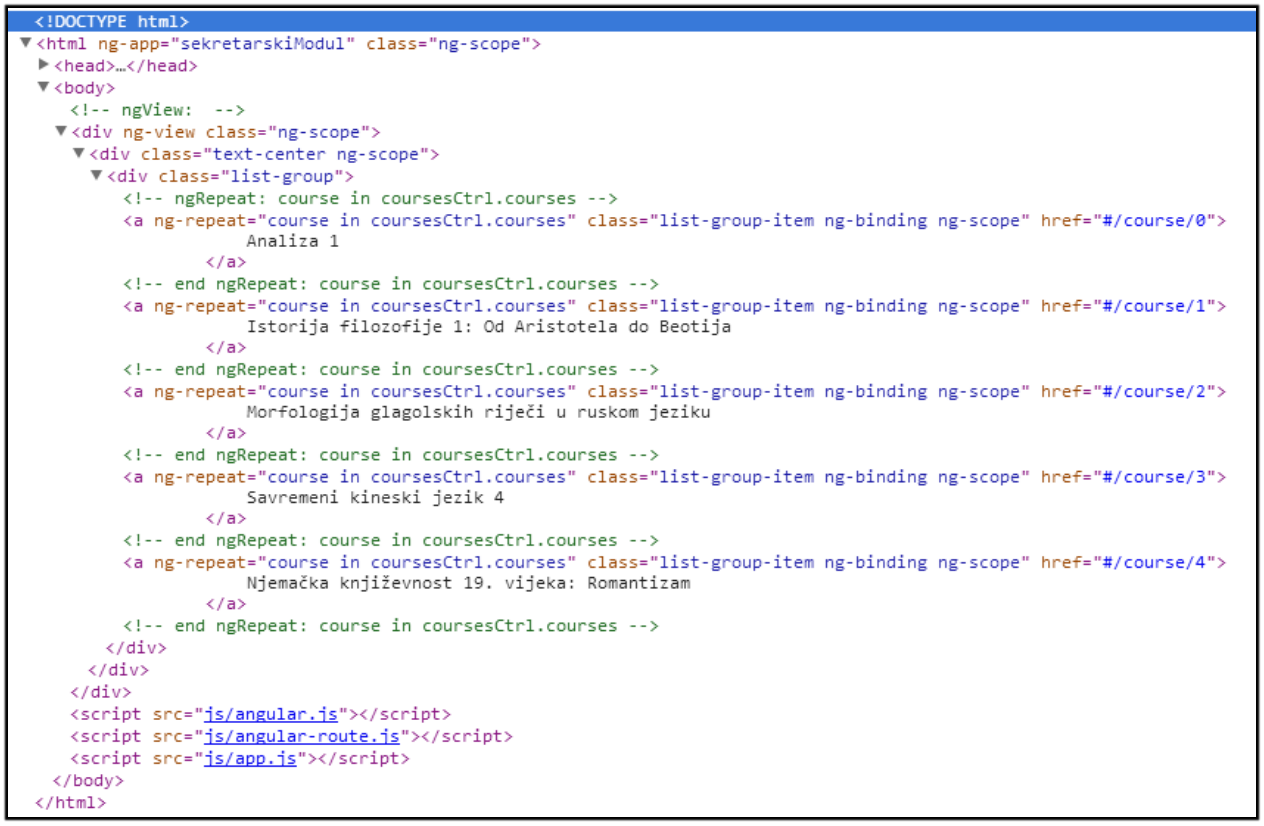
[85] Šablon u dizajnu softvera kojim se instanciranje klase ograničava na jedan jedini objekat.

```
        .error(function (data) {
            console.log(data);
        });
}]);
```

Definisani pogled će prikazivati listu predmeta, dobavljenih sa servera. Direktiva *ng-repeat*, kojom je označen *a* tag, dosta liči na *foreach* petlju koja ima svoje implementacije u mnogim programskim jezicima (*C++11, C#, Java, PHP, JavaScript...*). Naime, za svaki element niza (u ovom slučaju *courses* niza kontrolera) se generiše označeni tag (u ovom slučaju *a*), kao i svi njegovi potomci, a trenutnom elementu niza se može pristupiti pomoću pseudonima (u ovom slučaju *course*).

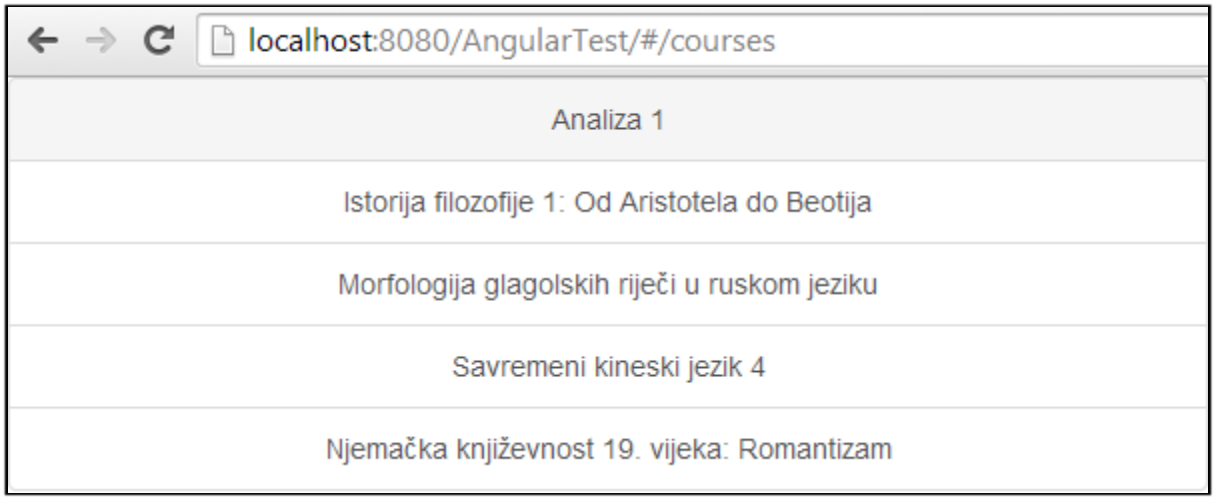
Unutar kontrolera se poziva *getAll* funkcija *CourseService* servisa, na koju se dodaju pozivi u slučaju uspješnog i neuspješnog *AJAX* poziva. U ovom slučaju, nakon uspješnog poziva će se dobijeni podaci proslijediti nizu *courses* kontrolera, a u slučaju greške prispjeli podaci ispisati u konzolu. Može se primjetiti uvođenje promjenljive *self*, u koju se smješta referenca za trenutni *this*, koji se odnosi na kontroler. Ovo je česta praksa pri konstrukciji *JavaScript* programa, zvog osetljivosti *this* reference u ovom jeziku. Ukratko, ovako je omogućen pristup članovima kontrolera i u drugim funkcijama, kao na primjer u onoj koja se poziva nakon uspješnog dobavljanja predmētā sa servera.

Nakon učitavanja radnog modula, izvršavanja koda i pristizanja podataka sa servera, rezultujući *HTML* će izgledati slično ovome:



Slika 5 – HTML generisan nakon dobavljanja predmeta sa servera

Može se primjetiti da je *AngularJS* generisao *a* tagove za sve prispjele predmete i formatirao *href* atribut i tekst unutar generisanog taga, u zavisnosti od dostupnih podataka. Takođe, *AngularJS* je *HTML* popunio dodatnim pomoćnim klasama i komentarima, koji služe za pravilno interno funkcionisanje aplikacije i lakše snalaženje pri čitanju *HTML*-a. Generisani *HTML* će se u pregledaču prikazivati na sljedeći način:



Slika 6 – Generisana lista predmeta

Sljedećim kodom će biti konstruisan radni modul za prikaz pojedinačnog predmeta.

app/views/course.html

```
<div class="text-center" ng-show="courseCtrl.errorMessage.length == 0">
    <table class="table table-bordered">
        <tbody>
            <tr>
                <td>Naziv</td>
                <td>{{ courseCtrl.course.Name }}</td>
            </tr>
            <tr>
                <td>Broj predavanja</td>
                <td>{{ courseCtrl.course.NumLectures }}</td>
            </tr>
            <tr>
                <td>Broj vježbi</td>
                <td>{{ courseCtrl.course.NumExercises }}</td>
            </tr>
            <tr>
                <td>Semestar</td>
                <td>{{ courseCtrl.course.Semester }}</td>
            </tr>
            <tr>
                <td>Studijski program</td>
                <td>{{ courseCtrl.course.StudyProgramName }}</td>
            </tr>
        </tbody>
    </table>
</div>
<div class="alert alert-danger" ng-hide="courseCtrl.errorMessage.length == 0">
    {{ courseCtrl.errorMessage }}
    <a href="#/courses" class="btn btn-xs btn-default pull-right">
```

```
        Nazad na predmete
    </a>
</div>
```

app/main.js

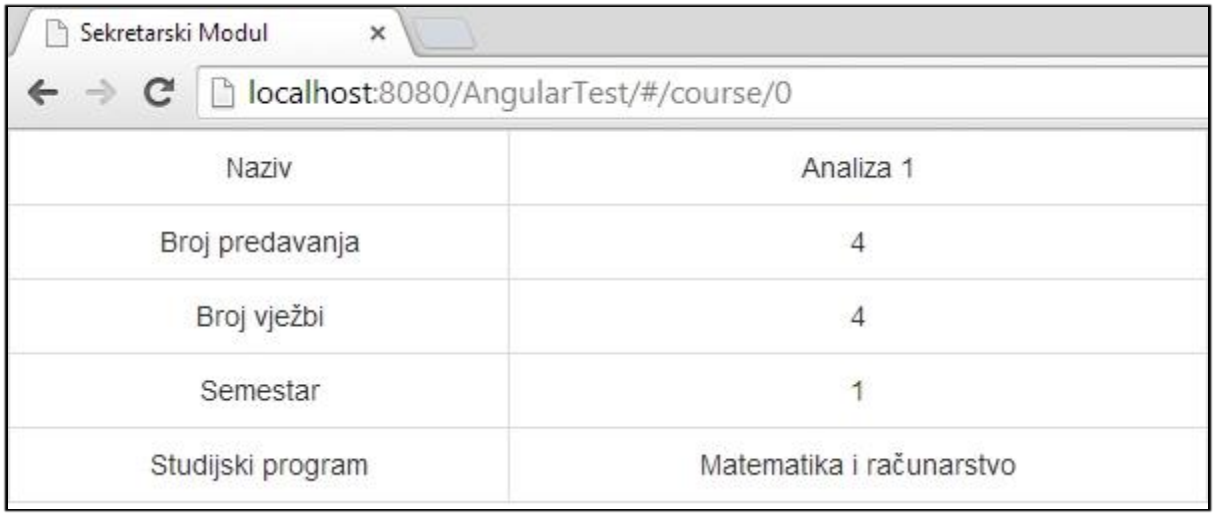
```
...

app.controller('CourseController', ['CourseService', '$routeParams',
    function (CourseService, $routeParams) {
        var self = this;
        self.course = {};
        self.errorMessage = '';
        CourseService.getById($routeParams.id)
            .success(function (data) {
                self.course = data;
            })
            .error(function (data) {
                self.errorMessage = data;
            });
    }
]);
```

Konstruisani pogled se sastoji od dva *div* taga, jednog koji će biti prikazan jedino ukoliko ne postoji poruka o grešci, a drugi koji će jedino tada biti sakriven. Pomenuto se ostvaruje pomoću *ng-show* i *ng-hide* direktiva. Direktiva *ng-show* će učiniti označeni tag vidljivim samo ukoliko je ispunjen uslov koji joj je proslijeđen (u ovom slučaju da je dužina poruke o grešci nula). Direktiva *ng-hide* radi upravo obrnuto od *ng-show*.

Prvi *div* pogleda prikazuje tabelu sa detaljima učitanog predmeta, a drugi poruku o grešci, sa vezom za povratak na listu predmeta.

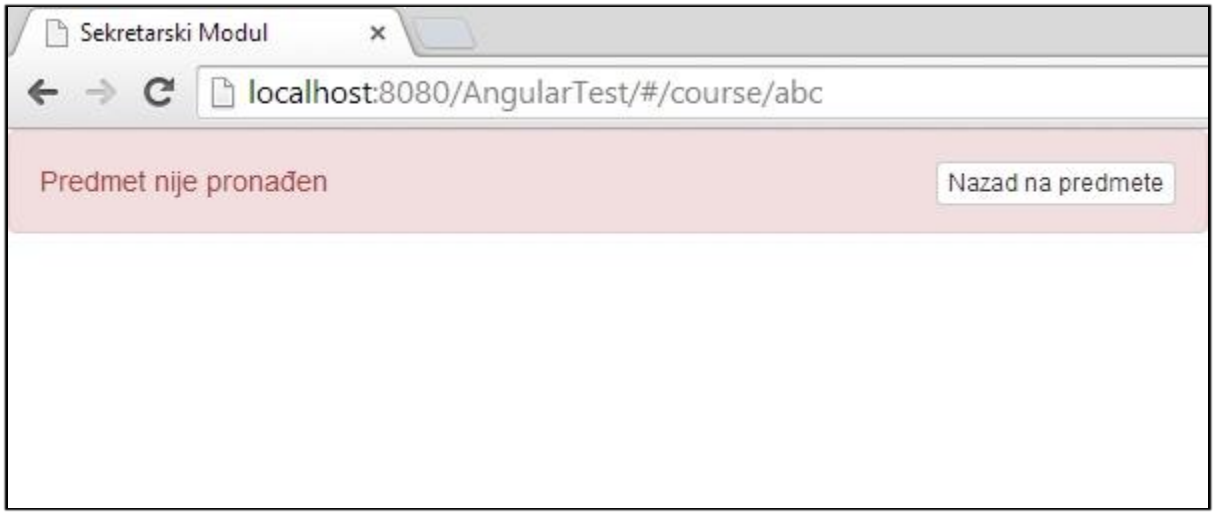
Kontroler koristi *CourseService* servis za dobavljanje podataka o predmetu i *$routeParams* servis za pristup parametrima rute. Pošto će ovaj radni modul biti korišten za rute oblika */course/:id*, koristi se *$routeParams* servis za pristup parametru *id* rute. Funkciji *getById* servisa *CourseService* se proslijeđuje *id* iz rute i dodaju pozivi za uspješno i neuspješno dobavljanje podataka. U slučaju greške, poruka o grešci će se sačuvati u *errorMessage* članu kontrolera. U suprotnom, podaci o predmetu se čuvaju u *course* članu kontrolera.

Nakon izvršavanja koda i učitavanja podataka, u pregledaču se prikazuje nešto slično sljedećem:



Slika 7 – Detalji o predmetu Analiza 1

Ukoliko se pokuša učitati nepostojeći poredmet, sljedeće se prikazuje u pregledaču:



Slika 8 – Poruka o grešci pri pokušaju učitavanja nepostojećeg predmeta

Prikazana aplikacija pobliže ilustruje SPA koncept. Glavni modul organizuje strukturu aplikacije i stara se o njenom globalnom funkcionisanju, dok kontroleri i servisi čine manje, radne i pomoćne module, koji obavljaju konkretnu poslovnu logiku i komuniciraju sa serverom.

6.4.2. Kompleksniji *Durandal* primjer

Durandal aplikacija u primjeru koji slijedi će izgledom i funkcionalnošću biti indentična prethodnoj *AngularJS* aplikaciji.

Views/SekretarskiModul/Index.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Sekretarski Modul - Durandal</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
                                 bootstrap/3.2.0/css/bootstrap.min.css">
    <style>
        body { max-width: 600px; }
    </style>
</head>
<body>
    <div id="applicationHost"></div>
    <script src="lib/require/require.js" data-main="app/main"></script>
</body>
</html>
```

Sadržaj datoteke *index.html* nije mnogo izmjenjen u odnosu na osnovni *Durnadal* primjer, dodat je samo *Twitter Bootstrap* razvojni okvir i jedan red *CSS* koda za stilizaciju, istovjetno kao u *AngularJS* aplikaciji.

app/main.js

```
requirejs.config({
    paths: {
        'text': '../lib/require/text',
        'durandal': '../lib/durandal/js',
        'plugins': '../lib/durandal/js/plugins',
        'knockout': '../lib/knockout/knockout-3.1.0',
        'jquery': '../lib/jquery/jquery-1.9.1',
        'services': './services'
    }
});

define(['durandal/app', 'durandal/viewLocator'],
    function (app, viewLocator) {
        app.configurePlugins({
            router: true
        });
```

```
        app.start().then(function () {
            viewLocator.useConvention();
            app.setRoot('viewmodels/shell');
        });
    }
);
```

Glavna skripta, *main.js*, takođe nije mnogo promjenjena. Pored prijašnjih, dodate su i prečice za *Durandal*-ove dodatke (*'plugins'*) i korisnički definisane servise (*'services'*). Od dodataka za *Durandal* u ovom primjeru će biti korišten *router*, što je navedeno prije pokretanja aplikacije. Nakon pokretanja, kao i u osnovnom primjeru, za korjeni modul se postavlja ljuska.

Slično *AngularJS* aplikaciji, glavni modul će konfigurisati rute dostupne u aplikaciji i mjesto u *HTML* dokumentu gdje će radni moduli iscrtavati korisnički interfejs.

app/views/shell.html

```
<div data-bind="router: {}"></div>
```

app/viewmodels/shell.js

```
define(['plugins/router'], function (router) {
    var vm = {
        activate: function () {
            router
                .map([
                    {
                        route: 'home',
                        moduleId: 'viewmodels/home'
                    },
                    {
                        route: 'courses',
                        moduleId: 'viewmodels/courses'
                    },
                    {
                        route: 'course/:id',
                        moduleId: 'viewmodels/course'
                    }
                ])
                .buildNavigationModel()
                .mapUnknownRoutes('viewmodels/home', 'home')
                .activate();
        }
    };
    return vm;
});
```

Skromni glavni pogled definiše *div* tag, gdje će *router* prikazivati pogled trenutno aktivnog radnog modula. Ovo se označava pomoću *data-bind* atributa i *router binding*-a, kome se mogu proslijediti dodatne konfiguracije, nepotrebne u ovom slučaju.

Svaki Durandal *VM* sadrži *activate* funkciju, koja se poziva odmah po aktiviranju modula. Ukoliko se ne definiše eksplicitno, razvojni okvir modulu dodaje praznu *activate* funkciju. U *activate* funkciji glavnog modula će biti izvršena konfiguracija ruta, koristeći *router* dodatak za *Durandal*, kome pristupamo pomoću *'plugins'* skraćenice.

Funkciji *map router*-a se prosljeđuje niz objekata sa podacima o rutama: fragmentnim djelom *URL*-a kojem odgovaraju i putem do *VM*-a modula. Nakon toga, potrebno je izgraditi navigacioni model pozivom na *buildNavigationModel*, postarati se za sve ostale rute pozivom na *mapUnknownRoutes* i aktivirati *router*. U navedenom primjeru se sve nepoznate rute preusmjeravaju na *viewmodels/home* modul i *home URL* fragment.

Slično kao u *AngularJS* aplikaciji, *URL* fragmentu *home* će odgovarati modul čiji je *VM* *viewmodels/home* modul, fragmentu *courses* modul *viewmodels/courses*, a fragmentu oblika *course/:id* modul *viewmodels/course*, s tim što će *:id* biti proizvoljan parametar rute, koji ne sadrži *'/'* karakter, analogno *AngularJS* aplikaciji.

AngularJS dozvoljava proizvoljno uparivanje pogleda i kontrolera, pa je pri definiciji rute neophodno eksplicitno navesti i kontroler i pogled koji joj odgovaraju. Za razliku od njega, *Durandal* koristi *MVVM* pristup, koji dozvoljava isključivo „1 na 1“ veze između pogleda i *VM*. Stoga je pri definiciji rute potrebno samo navesti put do *VM*-a, a *viewLocator* pronalazi odgovarajući pogled po konvenciji, kao i u osnovnom primjeru.

app/views/home.html

```
<div class="jumbotron text-center">
    <h1 data-bind="text: greeting"></h1>
    <a class="btn btn-lg btn-primary" href="#/courses">Predmeti</a>
</div>
```

app/viewmodels/home.js

```
define(['knockout'], function (ko) {
    var greeting = ko.observable('Dobrodošli!');
    var vm = {
        greeting: greeting
    };
```

```
    return vm;
});
```

Analogno *AngularJS* aplikaciji, početni modul prikazuje pozdrav korisniku i vezu prema listi predmeta. Za rad ostala dva modula je neophodan servis za dobavljanje podataka o predmetima, koji će biti sljedeći konstruisan:

app/services/coursesService.js

```
define(['jquery'], function ($) {
    var CourseService = {
        getAll: function () {
            return $.ajax({
                type: 'GET',
                url: 'Data/Courses'
            });
        },
        getById: function (id) {
            return $.ajax({
                type: 'GET',
                url: 'Data/Course/' + id
            });
        }
    };

    return CourseService;
});
```

Pomoćni servis je konstruisan kao zaseban *RequireJS* modul, koje se može proslijediti drugim modulima pomoću *DI*. Funkcionalnost konstruisanog modula je dosta slična pomoćnom servisu u *AngularJS* primjeru, s tim što za *AJAX* komunikaciju sa serverom koristi *jQuery.ajax* funkciju.

app/views/courses.html

```
<div class="text-center">
    <div class="list-group" data-bind="foreach: courses">
        <a data-bind="text: Name, attr: { href: '#/course/' + Id }"
           class="list-group-item">
        </a>
    </div>
</div>
```

app/viewmodels/courses.js

```
define(['knockout', 'services/coursesService'], function (ko, CoursesService) {
    var courses = ko.observableArray([]);
    var activate = function () {
        CoursesService.getAll()
            .done(function (data) {
                courses(data);
            })
            .fail(function (data) {
                console.log(data);
            });
    };

    var vm = {
        courses: courses,
        activate: activate
    };

    return vm;
});
```

Po pokretanju *viewmodels/courses* modula, pokreće se *getAll* funkcija *CoursesService* modula, na šta se dodaju pozivi nakon uspešne i neuspješne *AJAX* komunikacije. Ekvivalentno *AngularJS* aplikaciji, nakon uspješog dobavljanja podataka, oni se smještaju u *courses KnockoutJS* nadgledani niz (*observableArray*). Nadgledani niz predstavlja nadgledanu varijablu koja je niz i kojoj se pored obične promjene vrijednosti posebno prate i događaji dodavanja, uklanjanja i promjene redoslijeda elemenata.

U pogledu se pomoću *foreach* povezivanja postiže slično kao sa *AngularJS*-ovom *ng-repeat* direktivom, s tim što *foreach* za svaki element u nizu generiše samo potomke označenog taga. Stoga, potrebno je označiti *div* tag roditelj taga *a*, a ne sâm tag *a* kao u *AngularJS* primjeru. Trenutni element u nizu postaje glavni opseg imena za generisani *HTML*, pa ukoliko se navede *Id* ili *Name*, to će se odnositi na *id* i ime trenutnog predmeta u nizu. Stoga se tekst generisnog *a* taga može podesiti *text binding*-om, a *href* pomoću *attr binding*-a.

app/views/course.html

```
<div class="text-center" data-bind="visible: errorMessage().length == 0">
    <table class="table table-bordered">
        <tbody>
            <tr>
                <td>Naziv</td>
                <td data-bind="text: course().Name"></td>
            </tr>
            <tr>
                <td>Broj predavanja</td>
                <td data-bind="text: course().NumLectures"></td>
            </tr>
            <tr>
                <td>Broj vježbi</td>
                <td data-bind="text: course().NumExercises"></td>
            </tr>
            <tr>
                <td>Semestar</td>
                <td data-bind="text: course().Semester"></td>
            </tr>
            <tr>
                <td>Studijski program</td>
                <td data-bind="text: course().StudyProgramName"></td>
            </tr>
        </tbody>
    </table>
</div>
<div class="alert alert-danger" data-bind="visible: errorMessage().length != 0">
    <span data-bind="text: errorMessage"></span>
    <a href="#/courses" class="btn btn-xs btn-default pull-right">
        Nazad na predmete
    </a>
</div>
```

Analogno *AngularJS* aplikaciji, pogled za *viewmodel/course* modul se sastoji iz dvije sekcije. Jedne koja je vidljiva samo ako je prazna poruka o grešci, a druga koja je jedino tad vidljiva. Ovo se postiže pomoću *visible* povezivanja.

app/viewmodels/course.js

```
define(['knockout', 'services/coursesService'], function (ko, CoursesService) {
    var course = ko.observable();
    var errorMessage = ko.observable();
    var activate = function (id) {
        course({});
        errorMessage('');
        CoursesService.getById(id)
            .done(function (data) {
                course(data);
            })
```

```
            .fail(function (data) {
                errorMessage(data);
            });
    };

    var vm = {
        course: course,
        errorMessage: errorMessage,
        activate: activate
    };

    return vm;
});
```

Funkciji *activate* radnog modula se kao argumenti proslijeđuju svi parametri rute, ukoliko postoje. U ovom slučaju, to je samo *id*. Po aktiviranju ovog modula vrši se inicijalizacija nadgledanih promjenljivih i pokušava dobaviti traženi predmet, koristeći *CoursesService*. Po neuspješno obavljenoj komunikaciji, poruka o grešci će biti spašena u *errorMessage* nadgledanoj promjenljivoj, a u suprotnom će se podaci o predmetu smjestiti u *course* nadgledanu promjenljivu.

Vrijednosti sačuvanoj u nadgledanoj promjenljivoj se može pristupiti ukoliko je pozovemo kao funkciju bez argumenata, što je demonstrirano u pogledu. Pozivanjem *course()* se pristupa objektu sačuvanom u toj nadgledanoj promjenljivoj, pa se *course().Name* odnosi na naziv predmeta dobavljenog sa servera.

7. Zaključak

Veb je veoma široka i plodna softverska platforma. Konstantni tehnološki napretci su omogućili rasprostranjenost bogatih, interaktivnih klijentskih aplikacija. SPA pristup konstrukciji Veb aplikacija predstavlja elegantno rješenje problema distribucije poslovne logike i rasterećenja servera, a klijentima pruža glatko i dinamično iskustvo interakcije sa aplikacijom. Korištenjem modernih *HTML*, *CSS* i *JavaScript* tehnologija, moguće je plasirati softverska rješenja za većinu savremenih računara.

U radu su prikazani ključni dijelovi arhitekture SPA, kao i konkretni primjeri njihove implementacije. Prikazana je serverska aplikacija, koja najvećim djelom služi kao omotač za sloj podataka. Pristup podacima sistema se realizuje pomoću *API* poziva, a svaka interakcija sa serverskom aplikacijom prolazi kroz sigurnosne i validacione provjere. Prikazana je klijentska aplikacija, u kojoj se implementira većina poslovne logike sistema i sva interakciju sa korisnikom, a sa serverskom aplikacijom komunicira isključivo asinhrono.

Od daljeg istraživanja o predstavljenoj temi, neka od najznačajnijih su problem efikasnog dobavljanja podataka od servera, sinhronizacija između podataka na serveru i klijentu, sinhronizacija podataka između klijenata, sigurnost klijentske aplikacije i optimizovanje stranice za sisteme pretrage Interneta.

8. Literatura

[1] L. Shklar i R. Rosen, Web Application Architecture: Principles, Protocols, and Practices, Chichester, England: John Wiley, 2003.

[2] M. Leinster, A Logic Named Joe, Baen Publishing Enterprises, 2005.

[3] G. Wolf, „The (Second Phase of The) Revolution Has Begun," *Wired 2.10,* 1994.

[4] M. K. Gray, „Web Growth Summary," MIT, 20. 6. 1996. [Na mreži]. Available: http://www.mit.edu/people/mkgray/net/web-growth-summary.html. [Poslednji pristup 7. 4. 2014.].

[5] B. Wallace, „The History of Web Design," Dashburst LLC., [Na mreži]. Available: http://dashburst.com/infographic/history-of-web-design/. [Poslednji pristup 12. 4. 2014.].

[6] „Total Number of Websites," Internet Live Stats, [Na mreži]. Available: http://www.internetlivestats.com/total-number-of-websites/. [Poslednji pristup 23. 4. 2014.].

[7] M. Galli, R. Soares i I. Oeschger, „Inner-browsing Extending the Browser Navigation Paradigm," Mozilla Developer Network, 16. 5. 2003. [Na mreži]. Available: https://developer.mozilla.org/en-US/docs/Inner-browsing_extending_the_browser_navigation_paradigm. [Poslednji pristup 15. 5. 2014.].

[8] T. Berners-Lee i M. Fischetti, Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor, San Francisco: Harper San Francisco, 1999.

[9] S. Jobs, „Thoughts on Flash," Apple Inc, 4. 2010. [Na mreži]. Available: https://www.apple.com/hotnews/thoughts-on-flash/. [Poslednji pristup 17. 5. 2014.].

[10] „The Evolution of the Web," [Na mreži]. Available: http://www.evolutionoftheweb.com/. [Poslednji pristup 12. 4. 2014.].

[11] „Statistics - International Telecommunications Union," [Na mreži]. Available: http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2014/ITU_Key_2005-2014_ICT_data.xls. [Poslednji pristup 2. 7. 2014.].

[12] „Global Internet, Mobile and Social Media Engagement and Usage Stats and Facts,“ Social Media Today LLC, 11. 12. 2013. [Na mreži]. Available: http://socialmediatoday.com/irfan-ahmad/1993606/global-overview-internet-mobile-and-social-media-engagement-and-usage-infographi. [Poslednji pristup 17. 5. 2014.].

[13] C. Smith, „40 Amazing Google Stats and Facts,“ DMR, 2. 2. 2014. [Na mreži]. Available: http://expandedramblings.com/index.php/by-the-numbers-a-gigantic-list-of-google-stats-and-facts/. [Poslednji pristup 18. 5. 2014.].

[14] „How Much Email Do We Use Daily?,“ Source Digit RSS, 20. 2. 2014. [Na mreži]. Available: http://sourcedigit.com/4233-much-email-use-daily-182-9-billion-emails-sentreceived-per-day-worldwide/. [Poslednji pristup 18. 5. 2014.].

[15] „Skype - Microsoft Advertising,“ Microsoft Advertising, [Na mreži]. Available: http://advertising.microsoft.com/en/skype. [Poslednji pristup 18. 5. 2014.].

[16] „Smartphone Users Worldwide Will Total 1.75 Billion in 2014,“ EMarketer, 16. 1. 2014. [Na mreži]. Available: http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536. [Poslednji pristup 15. 5. 2014.].

[17] S. Smith, „Don't Repeat Yourself,“ O'Reilly, 24. 11. 2009. [Na mreži]. Available: http://programmer.97things.oreilly.com/wiki/index.php/Don't_Repeat_Yourself. [Poslednji pristup 20. 5. 2014.].

[18] S. Souders, „Velocity and the Bottom Line,“ O'Reilly, 1. 7. 2009. [Na mreži]. Available: http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html. [Poslednji pristup 23. 5. 2014.].

[19] M. S. Mikowski i J. C. Powell, Single Page Web Applications: JavaScript End-to-End, Shelter Island: Manning Publications, 2013.

[20] J. Papa, „SPA and the Single Page Myth,“ John Papa, Evangelist on the Loose, 30. 11. 2013. [Na mreži]. Available: http://www.johnpapa.net/pageinspa/. [Poslednji pristup 19. 5. 2014.].

[21] A. Martonik, „Larry Page: 1.5 Million Android Devices Activated Every Day,“ Mobile Nations, 18. 7. 2013. [Na mreži]. Available: http://www.androidcentral.com/larry-page-15-million-android-devices-activated-every-day. [Poslednji pristup 22. 5. 2014.].

[22] K.-M. Cutler, „Apple Has Sold 600M IOS Devices, But Android Is Not Impressed,“ AOL Inc., 10. 1. 2013. [Na mreži]. Available: http://techcrunch.com/2013/06/10/apple-android-2/. [Poslednji pristup 22. 5. 2014.].

[23] D. F. Pupius, „Rise of the SPA,“ Tech Talk, 6. 6. 2013. [Na mreži]. Available: https://medium.com/@dpup/rise-of-the-spa-fb44da86dc1f. [Poslednji pristup 23. 5. 2014.].

[24] T. Anglin, „HTML5: 10 Provocative Predictions For The Future,“ Say Media Inc., 22. 2. 2013. [Na mreži]. Available: http://readwrite.com/2013/02/22/html5-10-provocative-predictions-for-the-future. [Poslednji pristup 23. 5. 2014.].

[25] M. Boas, „The Future of Web Apps – Single Page Applications,“ The Worm Hole, 23. 8. 2010. [Na mreži]. Available: http://happyworm.com/blog/2010/08/23/the-future-of-web-apps-single-page-applications/. [Poslednji pristup 23. 5. 2014.].

[26] S. Brehm, „The Future of Web Apps Is -- Ready? -- Isomorphic JavaScript,“ VentureBeat, 8. 11. 2013. [Na mreži]. Available: http://venturebeat.com/2013/11/08/the-future-of-web-apps-is-ready-isomorphic-javascript/. [Poslednji pristup 23. 5. 2014.].

[27] C. Heilmann, „On Single Page Apps,“ Christian Heilmann RSS, 28. 12. 2011. [Na mreži]. Available: http://christianheilmann.com/2011/12/28/on-single-page-apps/. [Poslednji pristup 23. 5. 2014.].

[28] A. Kumar, „Single-Page Application (SPA) - Future Trends of Technology,“ C# Corner, 28. 4. 2014. [Na mreži]. Available: http://www.c-sharpcorner.com/Blogs/15347/single-page-application-spa-future-trends-of-technology.aspx. [Poslednji pristup 23. 5. 2014.].

[29] M. Takada, „Single Page Apps in Depth,“ [Na mreži]. Available: http://singlepageappbook.com/. [Poslednji pristup 30. 5. 2014.].

[30] M. Zgadzaj, „Benchmarking Node.js :: Change (b)log,“ ChangeBlog, [Na mreži]. Available: http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php. [Poslednji pristup 12. 09. 2014.].

[31] J. Costian, „Nginx vs Apache vs Express,“ 10. 11. 2013. [Na mreži]. Available: http://jamescostian.com/nginx-vs-apache-vs-express/. [Poslednji pristup 12. 09. 2014.].

[32] F. Frank, „Node.js Performance Case Study,“ 07. 12. 2011. [Na mreži]. Available: http://www.slideshare.net/FabianFrankDe/nodejs-performance-case-study. [Poslednji

pristup 12. 09. 2014.].

[33] J. Harrell, „Node.js at PayPal,“ PayPal, 13. 11. 2013. [Na mreži]. Available: https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/. [Poslednji pristup 12. 09. 2014.].

[34] A. Brice, „Is desktop software dead?,“ Successful Software, 28. 10. 2013. [Na mreži]. Available: http://successfulsoftware.net/2013/10/28/is-desktop-software-dead/. [Poslednji pristup 4. 7. 2014.].

[35] „RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax,“ The Internet Engineering Task Force, [Na mreži]. Available: http://tools.ietf.org/html/rfc3986#section-2. [Poslednji pristup 1. 9. 2014.].

[36] A. Lumsden, „A Brief History of the World Wide Web,“ Envato Pty Ltd., 25. 9. 2012. [Na mreži]. Available: http://webdesign.tutsplus.com/articles/a-brief-history-of-the-world-wide-web--webdesign-8710. [Poslednji pristup 2. 4. 2014.].

[37] World Wide Web Consortium, „About The World Wide Web,“ World Wide Web Consortium, 24. 1. 2001. [Na mreži]. Available: http://www.w3.org/WWW/. [Poslednji pristup 1. 7. 2014.].

[38] „History of the Web,“ World Wide Web Foundation, [Na mreži]. Available: http://webfoundation.org/about/vision/history-of-the-web/. [Poslednji pristup 2. 4. 2014.].

[39] M. Masnick, „Nanotech Excitement Boosts Wrong Stock,“ Techdirt, 4. 12. 2003. [Na mreži]. Available: https://www.techdirt.com/articles/20031204/0824235.shtml. [Poslednji pristup 22. 4. 2014.].

[40] „The History of the Web,“ W3C Wiki, 14. 3. 2014. [Na mreži]. Available: http://www.w3.org/wiki/The_history_of_the_Web. [Poslednji pristup 2. 4. 2014.].

[41] „AngularJS: Developer Guide,“ Google, [Na mreži]. Available: https://docs.angularjs.org/guide. [Poslednji pristup 23. 7. 2014.].

[42] „Docs | Durandal,“ Blue Spire, [Na mreži]. Available: http://durandaljs.com/docs.html. [Poslednji pristup 23. 7. 2014.].

9. Lista dijagrama i slika