Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matija Simin

Primerjava JavaScript ogrodij Angular, Backbone in Ember

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: viš. pred. dr. Aljaž Zrnec

Ljubljana 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V zadnjih nekaj letih je bilo razvitih več JavaScript MVC ogrodij. MVC poda aplikaciji strukturo in omogoča lažje obvladovanje, testiranje in ponovno uporabo kode. V okviru diplomskega dela najprej opišite zgodovino spletnega razvoja na strani odjemalca in predstavite MVC arhitekturo in njene različice. Opišite in primerjajte ogrodja AngularJS, Ember.js in Backbone.js glede na funkcionalnost, prilagodljivost, učinkovitost, testiranje, skupnost in dokumentacijo, ter predstavite implementacijo MVC arhitekture v vsakemu izmed njih. Razvijte manjšo spletno aplikacijo za praktični prikaz razvoja v izbranem ogrodju.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matija Simin, z vpisno številko **63080308**, sem avtor diplomskega dela z naslovom:

Primerjava JavaScript ogrodij Angular, Backbone in Ember

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom viš. pred. dr. Aljaža Zrneca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 2. marca 2016 Podpis avtorja:

Zahvaljujem se mentorju viš. pred. dr. Aljažu Zrnecu za njegovo pomoč in nasvete pri izdelavi diplomske naloge. Zahvaljujem se svoji družini za vso pomoč in razumevanje. Zahvaljujem se Ani za lektoriranje diplomske naloge.

Kazalo

Slike

Tabele

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	Vmesnik za programiranje aplikacij.
URL	Uniform Resource Locator	Enolični krajevnik vira.
JSON	JavaScript Object Notation	Notacija objektov Java Script, enostaven opisni jezik, ki ga razume JavaScript.
AJAX	Asynchronous JavaScript and XML	Asinhroni JavaScript in XML — tehnologija, ki omogoča osveževanje podatkov samo na delu strani.
DOM	Document Object Model	Objektni model dokumenta — programski API za HTML in XML dokumente. Definira logično strukturo dokumenta.
HTML	Hypertext Markup Language	Hipertekstovni označevalni jezik za opis spletnih dokumentov.
CSS	Cascading Style Sheet	Omogoča opis izgleda spletnega dokumenta.
XML	Extensible Markup Language	Označevalni jezik, ki definira zbirko pravil za kodiranje dokumentov. Narejen je bil za shranjevanje in prenos podatkov.

XSLT	Extensible Stylesheet Language Transformations	Jezik za pretvorbo XML dokumentov v druge XML dokumente.
SPA	Single-Page Application	So spletne aplikacije, ki naložijo eno HTML stran in jo dinamično osvežujejo glede na interakcijo z uporabnikom, brez konstantnih osveževanj strani.
GUI	Graphical user interface	Način, kako ljudje komunicirajo z računalnikom preko oken, ikon in menijev, ki jih lahko manipulirajo s pomočjo miške.
ORM	Object-relational mapping	Tehnika za dostop do podatkovne baze iz objektno usmerjenega programskega jezika.
MVC	Model-View-Controller	Arhitekturni model za implementacijo uporabniških vmesnikov, še posebej pa je popularen pri izdelovanju spletnih aplikacij.
SoC	Separation of Concerns	Je oblikovno vodilo za razdelitev računalniškega programa v več delov, ki opravljajo vsak svojo nalogo.

Povzetek

Diplomsko delo primerja tri JavaScript ogrodja za razvoj spletnih aplikacij, in sicer AngularJS, Ember.js in Backbone.js. V okviru dela smo primerjali njihovo implementacijo MVC arhitekturnega modela ali različice le-tega. Poleg implementacije MVC arhitekturnega modela smo primerjali njihove funkcionalnosti in lastnosti, kot so sistem predlog, implementacija usmerjevalnika, komunikacija s strežnikom, podatkovne povezave, prilagodljivosti, velikost in aktivnost skupnosti, kvaliteta obstoječe dokumentacije, možnosti za testiranje napisane kode, odvisnosti od zunanjih knjižnic za delovanje ter možnosti za razširitev funkcionalnosti ogrodja. V nadaljevanju diplomskega dela je nato predstavljen še razvoj testne aplikacije in uporaba nekaterih funkcionalnosti, ki jih ogrodje ponuja. V zaključku se poda mnenje, katero ogrodje izbrati.

Ključne besede: JavaScript, ogrodje AngularJS, ogrodje Ember.js, ogrodje Backbone.js, MVC

Abstract

The thesis compares three JavaScript frameworks for developing web applications: AngularJS, Ember.js and Backbone.js. In the thesis we compared their implementation of the MVC design pattern and their subversions. Besides their implementation of MVC design pattern we compared their functionalities and features, such as templating system, routing implementation, communications with a server, data binding, adjustability, size and activity of the community, quality of existing documentation, posibility of code testing, dependencies to external libraries and extensibility of the framework. In the remaining part of the thesis we describe development of a test application and the usage of some of their functionalities that the framework offers. In conclusion we give an opinion which framework to choose.

Keywords: JavaScript, AngularJS framework, Ember.js framework, Backbone.js framework, MVC

Poglavje 1

Uvod

V okviru izbire orodij, s katerimi bomo razvijali programsko opremo, je pomembnih veliko dejavnikov in velikokrat smo preplavljeni s količino informacij, mnenj in preferenc posameznikov ali skupin znotraj skupnosti. Če samo pogledamo na TodoMVC stran (dostopno na www.todomvc.com), ki nam pomaga pri izbiri, lahko zagledamo implementacijo TODO aplikacije v 19 različnih JavaScript knjižnicah. V času pisanja tega uvoda se je število lahko že povečalo. Najbolj pomembne lastnosti, ki pridejo na misel pri izbiri nove knjižnice, ogrodja ali orodja za razvoj programske opreme, je čas, ki ga potrebujemo za obvladovanje njihove zasnove, velikosti skupnosti, h kateri se lahko zatečemo po pomoč, izčrpnost dokumentacije in ažurnost popravkov hroščev, hitrosti razvoja in nenazadnje učinkovitosti izdelane rešitve z izbranim orodjem. Tukaj pa se vprašanja ne končajo, saj ta ogrodja ne obstajajo v izolaciji in je sodelovanje z drugimi obstoječimi tehnologijami prav tako pomembno, zato bomo v tej diplomski nalogi primerjali tri najbolj popularna, oziroma največja JavaScript MVC ogrodja.

Poglavje 2

Zgodovina razvoja na strani odjemalca

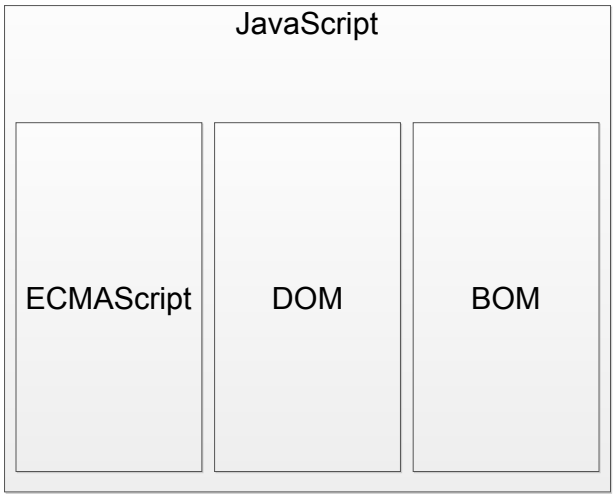
Programiranje na strani odjemalca se nanaša na programe, ki se izvajajo na strani odjemalca (npr. spletni brskalnik). To omogoča spletnim stranem, da se lahko odzovejo na uporabniška dejanja in vplive iz okolja. Skripte so lahko vključene v HTML dokument ali pa so prenesene iz strežnika v ločenem dokumentu. Po podatkih, ki so pridobljeni 28. januarja 2015 na spletni strani w3techs [21], je JavaScript glavni skriptni jezik za programiranje na strani odjemalca, saj je uporabljen v 88,2 odstotkih, sledi mu Flash z 11,9 odstotki, Java in Silverlight pa imata zanemarljive deleža uporabe. Obstajajo tudi drugi skriptni jeziki, kot so ActionScript, VBScript, Dart, TypeScript, Python.

2.1 JavaScript

JavaScript je visoko nivojski dinamičen programski jezik in je primeren za objektni in funkcionalni stil programiranja, pojavlja se v spletnih brskalnikih in omogoča izboljšano komunikacijo z uporabniki spletnih strani in aplikacij. Prvič se jezik pojavi leta 1995, njegova glavna naloga pa je bila preverjanje vnešenih podatkov v spletne obrazce, kar je bilo prej prepuščeno strežniškim jezikom. V času telefonskih modemov je bila vsaka zahteva, posredovana na strežnik draga in s tem, ko smo lahko preverili, ali je zahtevano polje prazno ali nepravilno izpolnjeno na strani

odjemalca, smo izključili veliko nepotrebnih obiskov strežnika.

Brendan Eich je začel z razvojem skriptnega jezika z imenom Mocha in kasneje LiveScript za Netscape Navigator 2 v letu 1995 [10]. Tik preden naj bi bil Netscape Navigator uradno izdan, je bilo ime jezika spremenjeno v JavaScript. Ker se je JavaScript tako dobro prijel, so se pri Microsoftu odločili, da izdajo svojo implementacijo JavaScripta, poimenovano Jscript. JavaScript 1.1, je bil leta 1997 poslan organizaciji Ecma (angl. European Computer Manufacturers Association) kot predlog. Tehnični odbor, sestavljen iz programerjev Netscap, Sun, Microsoft in drugih organizacij, je ustvaril standard ECMA-262, ki je definiral nov skriptni jezik imenovan ECMAScript [10]. Na sliki 2.1 lahko vidimo popolno implementacijo jezika JavaScript, ki je sestavljena iz treh delov.

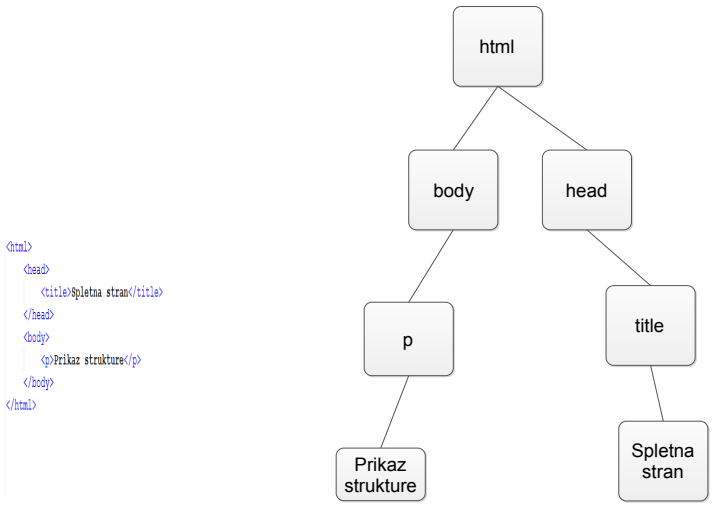


Slika 2.1: JavaScript implementacija.

Standard ECMA-262 definira ECMAScript jezik, ki služi kot osnova, na kateri lahko zgradimo bolj robustne skriptne jezike, kot sta JavaScript in Adobe Action Script. DOM in BOM predstavljata razširitev osnovnih funkcionalnosti ECMAScript, ki so vezane na okolje, v katerem se izvajajo (spletni brskalnik) [10].

DOM (angl. Document Object Model) je programski vmesnik za XML, ki je bil razširjen za uporabo v HTML. DOM predstavi spletno stran kot skupek vozlišč, ki so združena v tako imenovano DOM drevo. S tem, ko ustvarimo drevo,

ki predstavlja dokument, omogočamo razvijalcem nadzor nad vsebino in strukturo. Slika 2.2 prikazuje primer preproste spletne strani in njene DOM strukture.



Slika 2.2: DOM struktura.

Poznamo več stopenj DOM, in sicer:

- DOM 1. stopnje, ki je sestavljen iz dveh modulov:
  - Jedro DOM, ki ponuja način, kako strukturirati dokumente tipa XML za lahek dostop in manipulacijo katerega koli dela dokumenta.
  - DOM HTML, ki razširi jedro DOM s tem, ko doda metode in objekte za HTML dokumente.
- DOM 2. stopnje je dodal podporo za uporabniške dogodke, metode za potovanje čez DOM drevo in podporo za CSS (angl. Cascading Style Sheets). Vpeljal pa je tudi nove module:
  - DOM pogledi
  - DOM dogodki
  - DOM stil
  - DOM potovanje in razpon

- DOM 3. stopnje je vpeljal nove metode za nalaganje, shranjevanje in validacijo dokumentov.

BOM (angl. Browser Object Model) omogoča dostop do in manipulacijo brskalniškega okna. Razvijalci lahko komunicirajo z brskalnikom izven konteksta prikazane strani. BOM omogoča pridobivanje informacij o brskalniku, naloženi strani in nudi podporo za piškote itd. [10].

2.2 Ajax

AJAX (angl. asynchronous JavaScript and XML) je skupek tehnik za razvoj asinhronih spletnih aplikacij. Prvič se je izraz Ajax pojavil leta 2005, vendar pa so se tehnike, kjer ni bilo potrebno naložiti celotne spletne strani ob vsaki novi zahtevi, pojavile mnogo prej. Netscape Navigator 2.0 je bil prvi brskalnik, ki je podprl okvirje in JavaScript, in to še preden so bili okvirji uradno predstavljeni v HTML 4.0. To je pomenilo, da je bilo možno razdeliti spletno stran v več različnih dokumentov, ki so lahko pošiljali ločene zahteve na strežnik. Te posamezne okvirje in njihovo vsebino je bilo možno nadzirati z JavaScriptom.

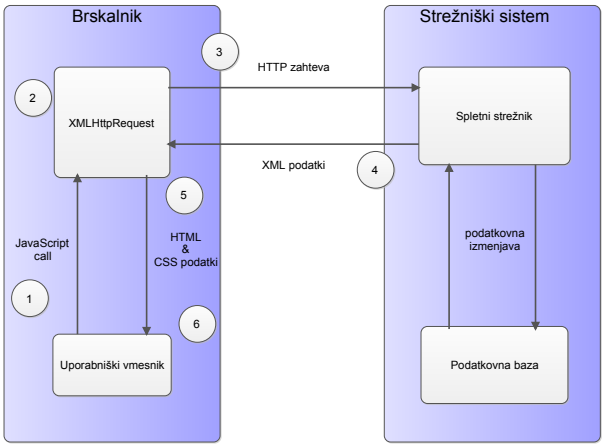
Pojavila se je tako imenovana tehnika skritih okvirjev, kjer je imel en okvir nastavljeno širino in višino na nič pikslov. Njegova glavna naloga je bila začeti komunikacijo s strežnikom. Skriti okvir je vseboval HTML obrazec s polji, ki so bila lahko dinamično izpolnjena s strani JavaScripta in poslana strežniku. Ko je okvir dobil odgovor od strežnika, je bila klicana JavaScript funkcija, ki je obvestila stran, ki je zahtevo sprožila, da so se zahtevani podatki vrnili. Ta tehnika je predstavljala prvi Ajax komunikacijski model.

Element <iframe> je bil predstavljen kot del HTML 4.0 v letu 1997. Razvijalci so lahko postavili <iframe> elemente kjerkoli na strani in s tem omogočili odjemalec — strežnik komunikacijo. Ko je bil DOM implementiran v Internet Explorer 5 in Netscape 6, so razvijalci lahko s pomočjo JavaScripta dinamično ustvarili <iframe> element, poslali zahtevo na strežnik, pridobili odgovor brez, da bi vključili dodaten HTML na spletni strani.

Pri Microsoftu so opazili popularnost tehnik skritega okvirja in se v letu 2001 odločili za predstavitev novega orodja za komunikacijo odjemalec —- strežnik v obliki ActiveX objekta, ki so ga poimenovali XMLHttp. S pomočjo tega objekta

so imeli razvijalci dostop do HTTP statusnih kod in glav ter podatkov, ki so bili vrnjeni s strani strežnika. Vrnjeni podatki so bili lahko v obliki XML, HTML, JavaScript objektov oziroma v katerem koli formatu si je razvijalec zaželel. Tako je bilo možno dostopati do strežnika programsko preko JavaScripta, neodvisno od nalagalnega cikla spletne strani. Pri Mozilla projektu so začeli z razvojem svojega XMLHttp. Reproducirali so metode in lastnosti v brskalniški objekt XMLHttpRequest. S tem se je razširil razvoj vmesnikov z Ajax funkcionalnostjo in prisilil brskalnika, kot sta Safari in Opera, da ponudita podporo za XMLHttp.

Ajax se od tradicionalnega komunikacijskega modela v spletnih aplikacijah razlikuje v tem, da ponudi vmesni sloj. To je JavaScript objekt ali funkcija, ki je klicana, ko se potrebuje informacije iz strežnika. Zahteva je izvedena asinhrono, kar pomeni, da se koda izvede in ne čaka na odgovor pred nadaljevanjem. Strežnik, ki bi tradicionalno serviral HTML, slike, CSS ali JavaScript, je sedaj konfiguriran, da vrne podatke, ki jih lahko Ajax uporabi. Podatki so lahko v obliki navadnega teksta, XML ali kateregakoli podatkovnega formata, ki ga lahko Ajax razume in interpretira. Ko Ajax prejme odgovor strežnika, analizira podatke in naredi spremembe na uporabniškem vmesniku, glede na informacije, ki so mu bile dane. Ker proces vsebuje prenos manjše količine informacij kot pri tradicionalnem modelu spletnih aplikacij, so posodobitve na uporabniškem vmesniku hitrejše in uporabnik opravi svoje delo hitreje. Delovanje Ajax lahko vidimo na sliki 2.3.



Slika 2.3: Ajax.

Tehnologije, ki predstavljajo Ajax, so: HTML/XHTML, CSS, DOM, XML, XSLT, XMLHttp, JavaScript [11].

2.3 jQuery

jQuery je odprtokodna JavaScript knjižnica, ki olajša interakcijo med HTML dokumentom oziroma njegovo DOM strukturo in JavaScriptom. Omogoča funkcionalnosti, kot so manipulacija in potovanje po DOM drevesu, odzivanje na dogodke, ustvarjanje animacij in Ajax aplikacij s preprosto sintakso. Prva stabilna različica je bila izdana leta 2006. jQuery reši problem nekonsistentnosti med spletnimi brskalniki pri interpretaciji JavaScript kode. Obstaja pa še mnogo drugih JavaScript knjižnic, ki pomagajo olajšati različne JavaScript naloge, kot so Dojo Toolkit, Mootools, Prototype in še mnoge druge.

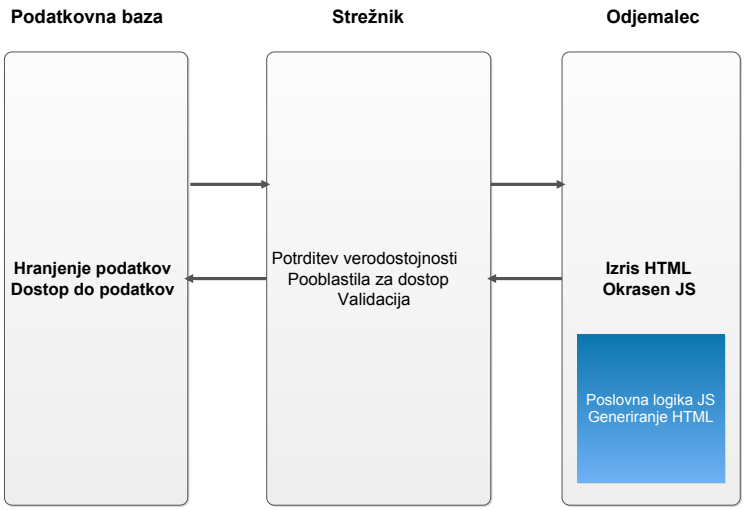
2.4 SPA

SPA (angl. Single-Page Application) je aplikacija, ki je dostavljena brskalniku in se med uporabo nikoli ponovno ne naloži. Vsi potrebni viri so naloženi ob prvem nalaganju. Glede na uporabniška dejanja pa se dodatni viri dinamično naložijo v ozadju. SPA dostavi namizne aplikacije v brskalnik. Obstaja več različnih SPA platform, kot so Java applets, Flash in JavaScript. Do leta 2000 sta se Flash in Java applets razvila v tej smeri, da je Java prinesla kompleksne aplikacije, kot je pisarniški paket v brskalniku; Flash pa je postal platforma za brskalniške igre in videa. Izdelava SPA aplikacij v JavaScriptu ponuja več različnih prednosti v primerjavi s prej omenjenima tehnologijama, kot so:

- Ni potrebe po ločenem vtičniku in varnostnih težavah, ki jih prinaša s seboj.
- Porabi manj virov kot vtičnik, ki potrebuje dodatno okolje za izvajanje.
- Uporaba samo enega jezika na odjemalcu.
- Bolj tekoče in odzivne spletne strani.

Slika 2.4 prikazuje strukturo SPA aplikacije [9]. V modrem okvirju so naloge, ki so bile v tradicionalnem modelu procesirane na strežniku, sedaj pa so prestavljene

na odjemalca.



Slika 2.4: SPA.

Poglavje 3

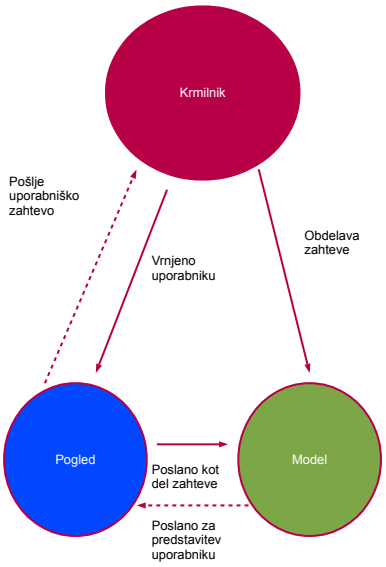
MVC (Model—View—Controller)

3.1 MVC

MVC je arhitekturni model, ki razdeli aplikacijo na tri dele. Organizira aplikacijo glede na arhitekturno načelo SoC (angl. Separation of Concerns), ki zapoveduje razdelitev računalniškega programa v več delov, le-ti imajo različne naloge. V primeru MVC arhitekturnega modela se izolira podatke aplikacije (Model) od uporabniškega vmesnika (Pogledi). Krmilniki upravljajo z logiko, uporabniškim vnosom in koordinacijo modelov ter pogledov.

Čeprav je bil model na začetku namenjen razvoju namiznih aplikacij, je postal eden glavnih modelov za razvoj spletnih aplikacij. Ustvarjeno je bilo veliko programskih ogrodji, ki so temeljila na tem modelu. Ta ogrodja se razlikujejo po svoji implementaciji, glede na razporeditev MVC dolžnosti med odjemalcem in strežnikom.

Zgodnja spletna MVC ogrodja so imela logiko modela, pogleda in krmilnika na strežniku. V tem pristopu odjemalec pošlje hiperpovezavo ali podatke iz spletnega obrazca do krmilnika in potem dobi nazaj celotno osveženo spletno stran. Model v celoti obstaja na strežniku. Z dozorevanjem tehnologij na strani odjemalca so bila ustvarjena MVC ogrodja, ki omogočajo, da se MVC komponente delno izvajajo na odjemalcu [15]. Slika 3.1 prikazuje preprosto implementacijo MVC arhitekture.



Slika 3.1: Prikaz MVC arhitekture.

3.1.1 Model

Model nudi funkcionalno jedro aplikacije. Zajame obnašanje aplikacije glede na problemsko domeno, ki jo obravnavamo (npr. banka, trgovina).

Vključi potrebne podatke in ponuja metode, ki izvajajo obdelavo teh podatkov, glede na potrebe aplikacije. Krmilniki kličejo te metode v imenu uporabnika. Model tudi ponuja metode za dostop do podatkov, ki so nato uporabljeni s strani pogledov. Ločitev modela od pogleda in krmilnika omogoča več različnih pogledov istega modela. Če uporabnik spremeni model skozi krmilnik enega pogleda, se mora sprememba poznati na vseh pogledih, ki so odvisni od teh podatkov. Model sporoči vsem pogledom, ko se njegovi podatki spremenijo. Pogledi nato pridobijo nove podatke od modela in posodobijo predstavitev le-teh [6].

3.1.2 Pogled

Pogled predstavi podatke uporabniku. Različni pogledi nam predstavijo podatke modela na različne načine.

Vsak pogled definira posodobitveno metodo, ki se aktivira s pomočjo meha-

nizma spremeni—razširi. Ko se posodobitvena metoda pokliče, pogled pridobi trenutne podatke za prikaz od modela in jih poda na zaslon. Med inicializacijo vseh pogledov so vsi pogledi povezani z modelom in registrirani z mehanizmom spremeni—razširi. Vsak pogled ustvari primeren krmilnik, med njima pa obstaja razmerje ena proti ena.

Pogosto pogledi ponujajo funkcionalnosti, ki omogočajo krmilnikom, da manipulirajo s prikazom. To je uporabno, ko gre za dogodke, ki ne vplivajo na model [6].

3.1.3 Krmilnik

Krmilnik sprejme uporabnikov vnos kot dogodek. Kako so ti dogodki podani krmilniku, je odvisno od uporabniškega vmesnika. Vsak krmilnik implementira metode, ki so klicane za vsak registriran dogodek. Dogodki so potem spremenjeni v zahteve za model in izbrani pogled. Če je obnašanje krmilnika pogojeno s stanjem modela, potem implementira posodobitveno metodo [6].

3.2 MVC različice

JavaScript MVC ogrodja se ne držijo popolnoma MVC arhitekture, zato pravimo, da ta ogrodja vpeljejo MV* model. Skozi čas se je MVC model razvil v več različnih implementacij, kot so MVP, MVVM, MVA. Vsem tem implementacijam je skupno, da združijo krmilnik s pogledom ali dodajo nove komponente namesto krmilnika [15]. Opisali bomo dve implementaciji, ki se pojavita pri obravnavanih JavaScript ogrodjih.

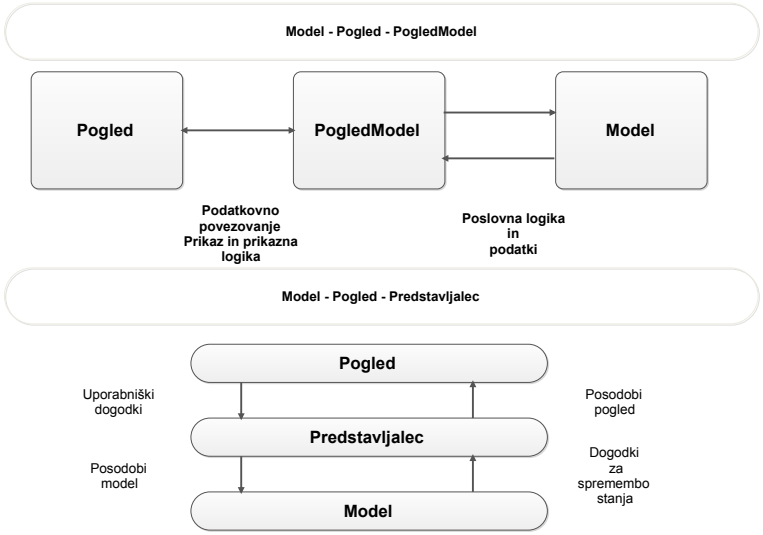
- MVP (Model—View—Presenter):

  MVP se razlikuje v tem, da krmilnik zamenja s predstavljalcem. Osredotoči se na izboljšavo predstavitvene logike. Predstavljalec je komponenta, ki vsebuje logiko uporabniškega vmesnika za dotični pogled. Predstavljalec zavzame vlogo mediatorja, ki komunicira tako s pogledom kot modelom. Obravnava uporabniške zahteve, pridobi podatke in določi, kako naj bodo prikazani v pogledu. Glavna prednost te implementacije je boljša razdelitev med pogledom in modelom ter lažje oziroma boljše testiranje [15].

- MVVM (Model—View—ViewModel):

  PogledModel manipulira z modelom in sproži dogodke v pogledu samem ter razkrije metode, komande in lasnosti, ki pomagajo vzdrževati stanje pogleda. Pogled se poveže na lasnosti PogledModela, ki nato razkrije podatke, ki jih vsebujejo objekti modela, ter stanja, ki so specifična za pogled [23].

Na sliki 3.2 lahko vidimo konceptualno predstavitev obeh modelov.



Slika 3.2: Prikaz MVP in MVVM arhitekture.

Poglavje 4

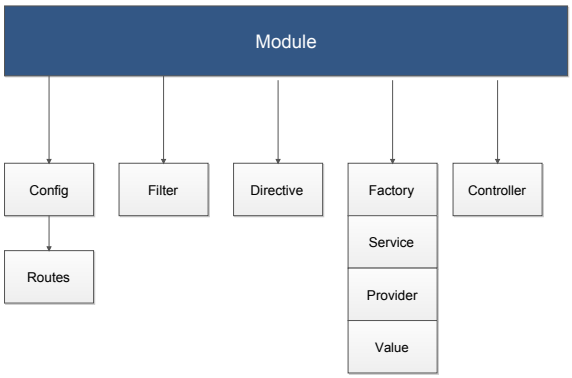
MVC ogrodja

4.1 AngularJS

AngularJS je odprtokodno JavaScript ogrodje in je podprto s strani Googla. Izvaja se v brskalniku in nam omogoča, da uveljavimo preverjene prakse, ki so bile že uporabljene v razvoju na strežniku, in s tem pospešimo razvoj. Primarno je uporabljen za razvoj SPA (angl. Single Page Application) aplikacij. Olajša izdelavo spletnih aplikacij s tem, ko doda nov nivo abstrakcije med razvijalcem in pogostimi nalogami, ki jih mora razvijalec implementirati v spletni aplikaciji. Ponuja nam različne funkcionalnosti, kot so [2]:

- razdelitev aplikacijske logike, podatkovnih modelov in pogledov.
- Ajax storitve.
- Dependency injection.
- zgodovina brskalnika.
- testiranje.

AngularJS podpira deklerativno programiranje pri sestavi uporabniškega vmesnika in imperativno programiranje za implementacijo poslovne logike aplikacije [12]. Na sliki 4.1 lahko vidimo prikaz komponent AngularJS.



Slika 4.1: Komponente AngularJS.

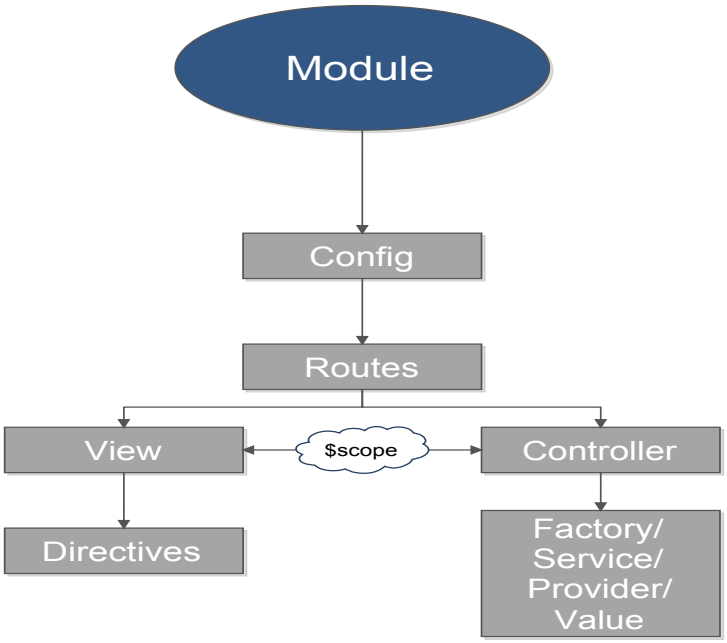
4.1.1 MVC v AngularJS

Scope je JavaScript objekt, ki lahko vsebuje podatke kot tudi funkcionalnosti. Preko scope objekta določimo, katere podatke modela in funkcionalnosti bomo razkrili pogledu za izris. Scope objekt predstavlja povezavo med krmilnikom in pogledom. Konceptualno so scopes zelo podobni PogledModelom iz MVVM.

Glavna naloga krmilnikov je inicializacija scope objektov. Vsebujejo poslovno logiko, pridobijo podatke in se odločijo, kateri del modela bodo razkrili pogledu. Krmilnik nima direktne reference na pogled, s tem pa je neodvisen od pogleda in se ga lažje testira.

Modeli so JavaScript objekti, podatki pa so predstavljeni z uporabo JSON objektov. Da jih predstavimo pogledu jih shranimo v scope objekt.

Pogledi oziroma predloge so napisane v HTMLju z uporabo direktiv, ki služijo kot razširitev HTML besednjaka [12]. Slika 4.2 prikazuje strukturo AngularJS in povezave med njenimi komponentami.



Slika 4.2: Prikaz strukture AngularJS.

Glavne naloge modula so definicija obsega in vstopne točke aplikacije ter organizacija kode in komponent. Modul omogoča enkapsulacijo kode. Olajša unit teste in deljenje kode med aplikacijami. V življenjskem ciklu modula obstajata dve fazi, in sicer konfiguracijska faza in faza izvajanja. Funkcija, ki je podana config metodi, se kliče, ko se naloži modul. V config metodi lahko registriramo storitve, vrednosti in konstante (niz, število, funkcija itd.) ter nastavimo API ključe [3].

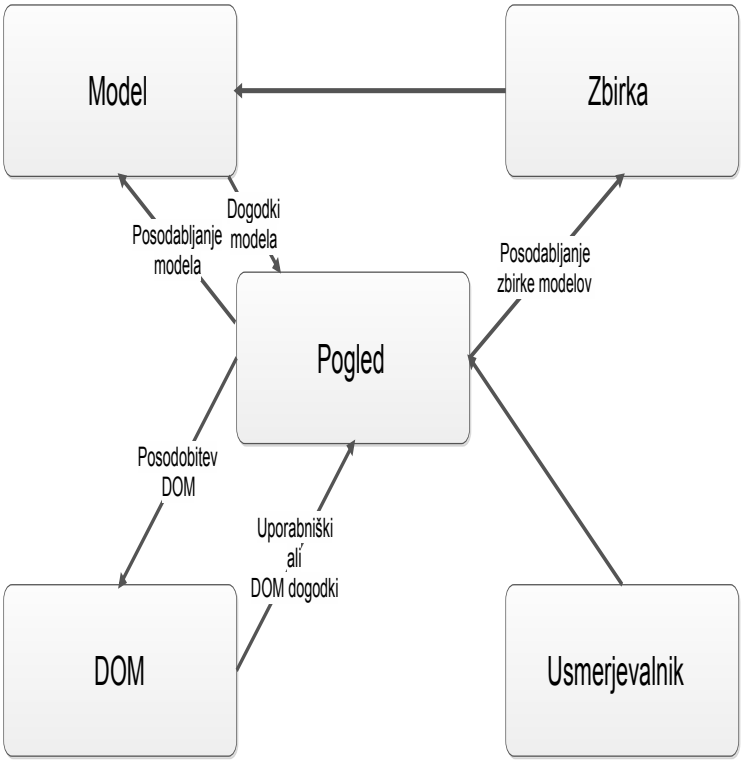
Storitve so uporabljene za enkapsulacijo funkcionalnosti, ki jo želimo uporabiti na več mestih v aplikaciji. So singleton objekti, ki so inicializirani samo enkrat. Če želimo uporabiti storitev, jo moramo prikazati kot odvisnost komponente, ki jo bo uporabila. To je lahko krmilnik, direktiva, filter ali katera druga storitev [1]. AngularJS ponuja veliko že vgrajenih storitev.

4.2 Backbone.js

Backbone.js je JavaScript ogrodje, ki se opira na MVC arhitekturo za strukturiranje aplikacij. Uporablja se za razvoj SPA aplikacij. Njena edina "trda" odvisnost je Underscore.js. Prva izdaja sega v leto 2010. Ne spada v t. i. "unopinionated" ogrodja, saj pusti veliko svobode pri ustvarjanju spletnih aplikacij in ponudi najbolj osnovne gradnike za strukturiranje podatkov, uporabniških vmesnikov ter navigacije [5].

4.2.1 MVC v Backbone.js

Slika 4.3 prikazuje implementacijo MVC modela v Backbone.js.



Slika 4.3: Prikaz strukture Backbone.js.

Backbone spada med MV* ogrodja. Backbone.Model (posamezni objekt) in

Backbone.Collection (skupina modelov) predstavljata model. Backbone.View predstavlja način, kako podati podatke iz modela v HTML. Backbone.Router določi, katera funkcija se bodo izvedla, ko uporabnik dostopa do določenega URLja v aplikaciji. Koncept krmilnika ni določen, vendar pa si njegove naloge med seboj razdelita pogled in usmerjevalnik [8].

Backbone se lahko najbolje opiše z MVP arhitekturo. Predstavljalec v MVP še najbolje predstavi Backbone.View. Pogled je povezan z modelom ali zbirko, manipulira z DOM in upravlja z dogodki, vezanimi na DOM strukturo. Pogled je lahko uporabljen kot klasičen pogled v MVC arhitekturi ali pa kot predstavljalec v MVP.

4.3 Ember.js

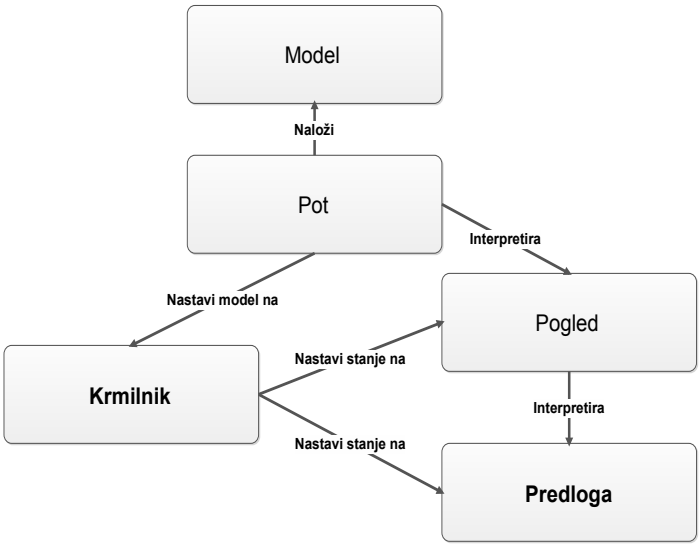
Ember.js je odprtokodno JavaScript ogrodje za izdelavo spletnih aplikacij, ki se izvajajo v brskalniku. Opira se na koncept "convention over configuration" in s tem zmanjša število odločitev, ki jih mora razvijalec sprejeti. Gradi na jQueryju in Handlebars kot knjižnici za izdelavo HTML predlog. Ogrodje ponuja dvosmerno podatkovno povezavo, mehanizme za navigacijo in upravljanje stanj aplikacije ter ORM (angl. Object—Relation Mapping) funkcionalnost na strani odjemalca [7].

Razvijalci Ember.js vidijo največjo moč spleta v zmožnosti zaznamka in deljenja URLjev, zato so združili orodja in koncepte GUI (angl. Graphical User Interface) ogrodji za podporo te funkcionalnosti. [16].

4.3.1 MVC v Ember.js

Preko uporabnika, ki sproži dogodek v obliki klika na povezavo, se sproži akcija, ki posodobi pot. Ta inicializira model in ga nastavi v krmilnik. Predloga dobi svoje podatke iz krmilnika izrisana pa je s strani poti.

Krmilniki predstavljajo vez med modelom in pogledom ali predlogo. Manipulirajo s podatki modela, ki jih dobijo iz poti; shranjujejo prehodne podatke; imajo definirane poslušalce za dogodke; ter preusmerijo dogodke, ki so namenjeni drugim krmilnikom, pogledom in predlogam. Omogočijo prehod z ene poti na drugo. Slika 4.4 prikazuje implementacijo MVC modela v Ember.js.



Slika 4.4: Prikaz strukture Ember.js.

Pogledi so povezani s krmilnikom, predlogo in s potjo. Uporabljajo se za implementacijo funkcij, ki se ne morejo implementirati v predlogah, in za ustvarjanje pogledov za večkratno uporabo.

Modele lahko pridobimo z uporabo AJAX klicev v definiciji poti, preko katere pridobimo JSON objekte, ki jih nato uporabimo kot modele. Vendar pa obstajajo višje nivojske rešitve v obliki knjižnic, kot je Ember Data, v kateri lahko definiramo modele, razmerja med njimi, ter ponuja kopico adapterjev in serializatorjev za komunikacijo s strežnikom [7].

Poglavje 5

Primerjava ogrodij

5.1 Funkcionalnosti

Vsa ogrodja ponujajo nekaj glavnih funkcionalnosti, ki nam olajšajo izdelavo SPA aplikacij, kot so pogledi, predloge, dogodki, modeli, komunikacija s strežnikom, podatkovne povezave in usmerjevalnik. V tem poglavju bomo predstavili, kako so te funkcionalnosti implementirane in kako se med seboj razlikujejo.

5.1.1 Sistem predlog

V ogrodjih, ki jih primerjamo, najdemo dva pristopa k procesiranju predlog. Pri prvem pristop je HTML interpretiran kot niz in iz tega niza ustvarjamo novo DOM drevo. Drugi pristop pa temelji na manipulaciji DOM drevesa s pomočjo direktiv v predlogah, s katerimi potem ustvarimo spremenjeno, dokončno DOM drevo, ki bo interpretirano v brskalniku.

AngularJS ima vgrajen svoj lastni deklerativen sistem predlog, ki temelji na DOM. Ko je HTML dokument sprejet, brskalnik začne z analizo in razčlenitvijo dokumenta, da lahko zgradi DOM drevo. Ko se gradnja drevesa zaključi, vstopi AngularJS prevajalnik in začne z iskanjem posebnih elementov, ki se imenujejo direktive.

Direktiva je razširitev HTML besednjaka. S to tehnologijo lahko ustvarimo komponente za večkratno uporabo. Direktiva je lahko uporabljena kot atribut, element, razred ali komentar. Angular vsebuje že svoje vgrajene direktive, lahko

pa ustvarimo tudi svoje [12]. Tabela 5.1 predstavlja nekaj najbolj pogosto uporabljenih direktiv, ki so vgrajene v AngularJS.

Direktiva	**Opis**
ngApp	Direktiva določi začetek in obseg aplikacije. Kot vrednost direktive podamo ime modula, ki služi kot vstopna točka aplikacije, na katero lahko potem priključimo krmilnike, storitve, filtre in direktive. V HTML dokumentu lahko obstaja samo ena direktiva ng-app.
ngController	S to direktivo lahko pripnemo določen krmilnik na pogled, s tem pa krmilnik in pogled začneta deliti isti scope objekt. Obstaja še en način, kako se pripne krmilnik na določen pogled, t. i. $route storitve.
ngBind	Ima isti pomen kot dvojni valoviti oklepaji za izraze.
ngRepeat	Se uporablja za potovanja čez zbirke in objekte. Ponuja pa še dodatne funkcionalnosti, s katerimi lahko, izvemo ali je index elementa prvi, zadnji, lih ali sod.
ngModel	Prilepi element na lastnost scopa, s tem pa poveže pogled z modelom.
ngClick	S to direktivo lahko definiramo obnašanje elementa, ko uporabnik klikne nanj. Obstajajo še druge direktive za določene dogodke, kot so ngBlur, ngChange, ngKeyPress itd.
ngDisable	Onemogoči element glede na resničnost podanega izraza.
ngClass	Uporabimo, ko želimo dinamično določiti CSS razred elementa glede na resničnost izraza.
ngOptions	Ustvari options elemente za HTML element select. Določimo, iz katere lastnosti scopa bo direktiva pridobila možnosti za izbiro. Za delovanje potrebuje ngModel direktivo.
ngStyle	Podoben koncept kot pri ngClass, vendar lahko tukaj direktno uporabimo lastnosti stila.
ngShow ngHide	Spremeni vidljivost elementa glede na vrednost njegove display lastnosti.

ngIf	Lahko določimo, ali se bo element izrisal ali ne.

Tabela 5.1: Tabela direktiv.

Na spodnjem primeru lahko vidimo uporabo ene izmed njih, in sicer ng-repeat.

```
<div ng-repeat="izdelek in izdelki">
   {{izdelek.cena}}
</div>
```

Kot je bilo omenjeno že v zgornjem odstavku, lahko ustvarimo tudi po svoji meri narejene direktive. Prvi korak pri izgradnji direktive je registracija v modul. Nato določimo vrednosti za lastnosti direktive, kot so templateUrl, replace, restrict, scope, transclude. V link funkciji lahko dostopamo do DOM, kjer lahko dodamo, kako se direktiva odziva na dogodke. Tabela 5.2 predstavlja lastnosti direktive in njihov opis.

Lastnosti direktive	**Opis**
templateUrl	Pot do HTML predloge za direktivo.
replace	Nastavimo, ko želimo, da se originalni element zamenja z vsebino predloge.
restrict	Direktivo lahko prikažemo kot atribut v elementu, element, razred ali komentar.
scope	Način, kako podamo podatke v direktivo.
transclude	Če želimo vstaviti del dokumenta v predlogo preko reference.
link	Če želimo dostopati do DOM in komunicirati z elementi, naj se to zgodi v link funkciji.
require	Nam omogoča, da dodamo krmilnik zunanje direktive, in s tem omogočimo komunikacijo med direktivami.

Tabela 5.2: Direktiva po meri.

Slika 5.1 prikazuje definicijo in strukturo direktive po meri.

Backbone ne predpisuje nobenega sistema predlog, zato imamo na voljo veliko možnosti. Lahko se odločimo za Underscore način grajenja predlog, kjer so

```
app.module("imeDirektive", function(){
    return{
        restrict: 'E',
        scope:{},
        templateUrl: 'template.html',
        replace: true,
        transclude: true,
        link: function(){
        }
    };
});
```

Slika 5.1: Predloga direktive.

predloge postavljene v HTML stran med script oznake s tipom text/template. Omogoča nam uporabo JavaScript izrazov, kot so zanke, pogojni stavki, ki jih podamo med <% %> , in spremenljivke med <%= %> oznake [8].

```
<script type="text/template" id="predloga">
        <ul>
                <% for (var i = 0; i<register.length; i++){ %>
                        <% var oseba = register[i]; %>
                        <li>
                                <%= oseba.ime %> <%= oseba.priimek %>
                        </li>
                <% } %>
        </ul>
</script>
```

Še ena izmed možnosti je Handlebars knjižnica, ki pa ne vsebuje JavaScript konstruktov v svojih predlogah, temveč zahteva uporabo že vgrajenih konstruktov. Na spodnjem prikazu se vidi uporabo le-teh.

```
<script type="text/x-handlebars-template" id="predloga">
        <ul>
```

```
            {{#each register }}
            <li> {{ime}} {{priimek}} </li>
            {{/each}}
        </ul>
</script>
```

Oba sistema predlog Underscore in Handlebars spadata v kategorijo, ki sestavi HTML iz podanega niza. Obstaja pa še veliko drugih možnosti pri izboru predložnih knjižnic, kot sta npr. Mustache in jQuery.tmpl.

Ember uporablja Handlebars.js za svoj sistem predlog. Handlebars predloge lahko vključimo direktno v HTML stran med script oznako tipa text/x-handlebars, lahko pa jih organiziramo v ločene datoteke s hbs končnico. Ponuja nam tudi možnost, da vključimo osnovno logiko in pravila formatiranja izpisa v naših predlogah.

Nova pridobitev na tem področju je HTMLBars, ki preide z manipulacije nizov na manipulacijo DOM vozlišč in je zgrajena na Handlebars. Ta pristop naj bi izboljšal učinkovitosti oz. hitrost pri upodobitvi dolgih seznamov in pomagal pri nekaterih varnostnih problemih, ki so prisotni pri manipulaciji nizov. Ni nam potrebno več uporabljati pomagače v obliki bind-attr kot poprej. Na spodnjem primeru lahko vidimo prikaz sintakse v HTMLBars.

```
<div class="{{mojRazred}}">
   {{izdelek.cena}}
</div>
```

Ember nam omogoča ustvarjanje komponent za večkratno uporabo, ki nam omogočajo, da enkapsuliramo strukturo in obnašanje. Novo komponento definiramo s tem, da ustvarimo novo predlogo, katero ime se začne s components. Komponente morajo v imenu vsebovati pomišljaj, npr. izdelek—podrobnosti.

```
script type="text/x-handlebars"
    id="components/izdelek-podrobnosti">
  //HTML za izpis podrobnosti
</script>
```

Če želimo definirati obnašanje komponente, to storimo z razširitvijo podrazreda Ember.Component [17].

```
Aplikacija.IzdelekComponent = Ember.Component.extend({
  //definicija lastnosti in akcij komponente
});
```

5.1.2 Usmerjevalnik

Vsa ogrodja imajo vgrajeno svojo implementacijo usmerjevalnika, ki nam omogoča navigacijo na strani odjemalca. Pri AngularJS in Backbone imamo na izbiro, ali uporabimo njihovo implementacijo usmerjevalnika ali zunanjo, npr. ui-router za AngularJS (ponuja dodatne funkcionalnosti) in npr. Marionette usmerjevalnik (ki razširi že ponujen usmerjevalnik s strani Backbone). Ember ne ponuja te izbire, vendar pa je vgrajen v ogrodje zelo zmogljiv usmerjevalnik, ki omogoča vgnezdene poti za razliko od ostalih dveh.

Če želimo v AngularJS aplikaciji omogočiti navigacijo, potrebujemo dodaten modul, in sicer ngRoutes. Nato moramo definirati ta modul kot odvisnost našega aplikacijskega modula. Naslednji del nam prikaže uporabo $routeProvider iz $route storitve za definiranje poti v aplikaciji:

```
aplikacija.config(['$routeProvider',
   function($routeProvider){
$routeProvider
     .when}('/', {
          templateUrl: 'pogledi/pogled.html',
          controller: 'Krmilnik.js'
     });
    }]);
```

Za definicijo poti uporabimo config metodo modula. Da dodamo specifično pot, uporabimo when() metodo, ki sprejeme dva parametra. Prvi je pot, drugi pa konfiguracijski objekt, ki mu lahko nastavimo stvari, kot so lokacija predloge, krmilnik itd. Obstaja še metoda otherwise(), ki se uporabi, ko se nobena pot ne ujema

s trenutnim URLjem [1]. V pogledu potem vključimo še ng-view direktivo, kjer določimo, kje v DOM hočemo, da se izriše predloga trenutne poti.

Backbone usmerjevalnik ponuja način, kako povezati URLje z različnimi deli aplikacije. V usmerjevalniku določimo funkcije, ki se sprožijo, ko uporabnik obišče določeno pot. Usmerjevalnik ustvarimo z Backbone.Router [5].

```
//definicija nove zbirke in poti v aplikaciji
Usmerjevalnik = Backbone.Router.extend({
        routes:{
                //dolocimo funkcijo, ki se aktivira ob obisku
                    dolocenega URLja
                'url':'imeFunkcije'
        },
        imeFunkcije = function(){}
});
//ustvarimo novo instanco usmerjevalnika
mojUsmerjevalnik = new Usmerjevalnik();

//da lahko spremljamo spremembe
Backbone.history.start();
```

Če želimo v aplikaciji URLje brez znaka # , lahko uporabimo HTML5 pushState. Usmerjevalnik nam omogoča definicijo poslušalcev za določene dogodke.

V Emberju se navigacija razdeli na dve fazi. Prva faza je izgradnja slovarja vseh poti oz. stanj v aplikaciji. To storimo z map funkcijo. Lahko pa tudi definiramo skupino poti z resource.

```
aplikacija.Router.map(function(){
        this.route('izdelek');
});
```

V zgornjem primeru Ember za nas zgenerira krmilnik, pot in predlogo v formatu z naslednjimi imeni: IzdelekController, IzdelekRoute in predlogo poimenovano izdelek. Poleg tega pa nam avtomatično zgenerira še ApplicationRoute in IndexRoute s pripadajočimi krmilniki in predlogami. Usmerjevalnik nam preko URLja pridobi

model, pogled, krmilnik in predlogo za nas. Če želimo spremeniti obnašanje poti, lahko to storimo s podrazredom Ember.Route. Preko poti lahko dostopamo do modela in ga tako ponudimo na razpolago krmilniku in predlogi. Poti so lahko uporabljene za nastavitev lastnosti krmilnika, izvajanje akcij in dogodkov [7].

```
aplikacija.IzdelekRoute = Ember.Route.extend({
        model: function(parametri){
                //AJAX klic za pridobitev modela
        }
});
```

5.1.3 Komunikacija s strežnikom

V SPA aplikacijah se komunikacija s strežnikom dogaja preko AJAX zahtev, ki omogočajo pridobivanje virov v obliki XML, JSON ali HTML brez osveževanja celotne spletne strani.

AngularJS ponuja dve glavni storitvi za pošiljanje oziroma izgradnjo in konfiguracijo Ajax zahtev. $http storitev je uporabljena za Ajax klice in je ovoj okrog brskalniškega XMLHttpRequest objekta. Funkcija sprejeme konfiguracijski objekt, ki je uporabljen za izvedbo HTTP zahteve in vrne obljubo, ki ima dve metodi za uspešno in neuspešno izvršeno zahtevo. Prikaz $http metode:

```
$http({
        method: 'GET',
        url: '/api/uporabniki'
}).success(function(data, status, headers, config){
        //klicano, ko je odgovor pripravljen
}).error(function(data, status, headers, config){
        //klicano, ko ima odgovor status napake
});
```

$http metoda vrne obljubo in potem lahko z then metodo pridobimo objekt odgovora. S $http storitvijo lahko pošiljamo (GET, POST, PUT, DELETE, JSONP) zahteve.

Če pridobivamo podatke iz RESTful APIja, uporabimo $resource storitev, pri tem pa moramo vključiti ngResource modul kot odvisnost za naš aplikacijski modul.

```
var uporabnik = $resource('/api/uporabniki/:id.json',
        {
                id:'@id'
        }
);
```

Storitev vrne objekt, ki ponuja get(), query(), save(), remove(), delete() metode.

Backbone predpostavlja, da komuniciramo s RESTful API, vendar pa omogoča, da sami nadomestimo metodo Backbone.sync() z drugačnimi nastavitvami. Ponuja preprost API, ki omogoča RESTful operacije na modelu in zbirki. To omogočimo s definiranjem url lastnosti v zbirki ali pa urlRoot lastnosti v modelu, če ga uporabljamo izven zbirke. Pridobivanje skupine modelov se dogaja na ravni zbirke s klicanjem fetch() funkcije, ki pošlje GET zahtevo in dobi odgovor v JSON formatu. Ustvarjanje novega modela v zbirko in shranjevanje na strežnik se lahko naredi za posamezni model s funkcijo save() ali pa na ravni zbirke s create(). Brisanje posameznega modela pa dosežemo z destroy() funkcijo [5].

```
//definicija nove zbirke
Zbirka = Backbone.Collection.extend({
        model: Izdelek,
        url: '/api/izdelki'
});
//ustvarimo novo instanco zbirke
mojaZbirka = new Zbirka();
//pridobimo vse izdelke
mojaZbirka.fetch();
```

V Ember lahko modele pridobimo z uporabo AJAX klicev v definiciji poti, preko katere pridobimo JSON objekte, ki jih nato uporabimo kot modele. Vendar pa obstajajo višje nivojske rešitve v obliki knjižnic, kot je Ember Data, v kateri lahko definiramo modele, razmerja med njimi, ter nam ponuja kopico adapterjev in

serializatorjev za komunikacijo s strežnikom, ni pa del ogrodja. Na voljo so še druge rešitve v obliki Ember Model, Ember Restless in Ember Persistence Foundation [7]. Na spodnjem primeru lahko vidimo definicijo modela in njegove atribute.

```
aplikacija.Izdelek = DS.Model.extend({
        naziv: DS.attr(),
        cena: DS.attr()
});
```

Lahko pa določimo tudi razmerja med več modeli, kot je prikazano na spodnjem primeru.

```
aplikacija.Narocilo = DS.Model.extend({
        izdelki: DS.hasMany('izdelek')
});

aplikacija.Narocilo = DS.Model.extend({
        narocilo: DS.belongsTo('narocilo')
});
```

V Ember Data je store centralno skladišče zapisov v aplikaciji. Store se uporablja za pridobivanje in ustvarjanje novih zapisov ter shranjevanje le-teh nazaj na strežnik. Na spodnjem primeru lahko vidimo primer iskanja določenega izdelka.

```
this.store.find('izdelek', 23);
```

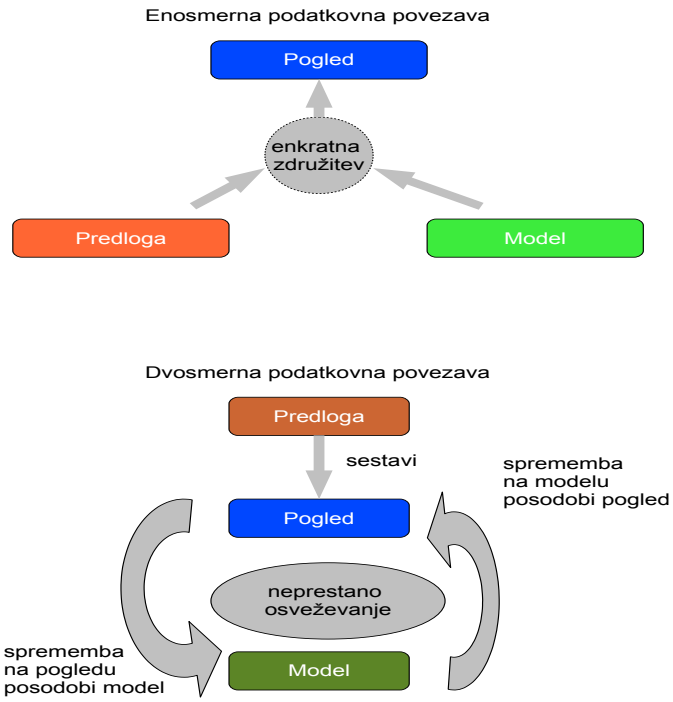
Store komunicira s strežnikom preko adapterja, ki pretvarja zahteve za pridobitev in shranjevanje zapisov v primerne klice strežniku [18].

5.1.4 Podatkovna povezava

Podatkovna povezava je proces vzpostavitve povezave med uporabniškim vmesnikom in poslovno logiko. Dvosmerna podatkovna povezava pomeni, da se s spremembo lastnosti modela spremeni posledično tudi uporabniški vmesnik; s spremembami elementov v uporabniškem vmesniku, so le-te vidne tudi v modelu. AngularJS in Ember ponujata avtomatično dvosmerno podatkovno povezavo, ko

Backbone nima te funkcionalnosti, vendar pa jo lahko sami implementiramo s pomočjo Backbone Events. Ta implementacija je primerna za preproste primere, v bolj kompleksnih situacijah pa se lahko spremeni v nepregledno zmešnjavo, zato je bolje uporabiti različne vtičnike za podporo le-tega npr., EPOXY.js ali Model-Binder.

Podatkovna povezava med modelom in pogledom se avtomatično sinhronizira v AngularJS aplikacijah. Ko se model spremeni, se to pozna v pogledu in obratno. Večina sistemov za predloge povezuje podatke samo v eno smer. Združijo predlogo in podatke v pogled, ko pride do spremembe v modelu ali pogledu, se to ne opazi takoj [14]. Slika 5.2 prikazuje razliko med dvosmerno in enosmerno podatkovno povezavo.



Slika 5.2: Prikaz dvosmerne in enosmerne podatkovne povezave.

S pomočjo ng-model direktive omogočimo dvosmerno podatkovno povezavo, kot je vidno na spodnjem primeru.

```
<body ng-app>
  Izdelek: <input type="text" ng-model="izdelek"/>
  Iskani izdelek: {{izdelek}}
</body>
```

Ember omogoča dvosmerno podatkovno povezavo s pomočjo "input helpers". Če so atributi elementa v narekovajih, se njihove vrednosti nastavijo na element, v nasprotnem primeru pa so povezane na lastnost v predlogi [19].

```
Izdelek: {{input type="text" value=izdelek}}
Iskani izdelek: {{izdelek}}
```

V Backbone lahko to funkcionalnost dosežemo z vključitvijo različnih knjižnic. V tem primeru bomo prikazali rešitev EPOXY.js, ki ponuja razširitve za komponente Model in View Backbona, da dosežemo dvosmerno podatkovno povezavo.

```
var Pogled = Backbone.Epoxy.View.extend({
      el: "#element",
      bindings:{
         //Dolocimo povezave in dogodke med pogledom in atributi
         //modela
      }
});
```

5.2 Prilagodljivost

Prilagodljivost ogrodja je odvisna od več dejavnikov, ki lahko na eni strani ponudijo veliko svobode razvijalcu pri strukturi aplikacije in implementaciji funkcionalnosti, ter izbiri orodji, ki jih bo pri tem uporabil. Na drugi strani pa lahko ogrodje vsili svojo miselnost o strukturi in implementaciji določenih funkcionalnosti.

Backbone je na tem področju najbolj prilagodljiv, saj prepušča veliko odločitev razvijalcu pri strukturiranju svoje aplikacije. Je zelo majhne velikosti, dobro sodeluje z drugimi knjižnicami in tehnologijami ter se ga lahko vključi na majhen del spletne strani. Dodatne funkcionalnosti, ki jih ogrodje ne ponuja, pa se lahko

implementirajo preko drugih knjižnic in ogrodji, kot so Marionette, Chaplin, Thorax.

AngularJS ni tako prilagodljiv kot Backbone, saj ponuja svojo implementacijo dvosmerne podatkovne povezave. Ima predpisan sistem predlog v obliki direktiv, preko katerih izrazimo funkcionalnosti aplikacij. Predpisuje modularno strukturo aplikaciji in ima svoj sistem za obvladovanje odvisnosti med komponentami. Vodi nas v to, da je testiranje čim lažje. Podpira deklarativen pristop k programiranju. Želi obdržati DOM manipulacijo v direktivah in ponuja svojo verzijo jQuery, imenovano jQLite. AngularJS aplikacijo se lahko vključi v že obstoječo aplikacijo, meje pa določimo s ng-app direktivo.

Ember prevzame nadzor nad celotno spletno stranjo. Deluje na principu "convention over configuration", ki od razvijalca zahteva, da sam določi obnašanje določenih komponent samo v primeru, če odstopajo od osnovnih smernic, vse ostalo pa za njega ustvari ogrodje. Tako kot AngularJS ima tudi Ember svojo implementacijo dvosmerne podatkovne povezave, svoj sistem za obvladovanje odvisnosti med komponentami in predpisuje svoj sistem predlog z Handlebars. Ember ponuja način za izgradnjo svojih HTML oznak za večkratno uporabo v obliki Components.

5.3 Dokumentacija in skupnost

Učenje novega ogrodja, njegove strukture in funkcionalnosti, ki jih ta ponuja, nam močno olajša dobra dokumentacija in obsežna skupnost.

AngularJS na svoji spletni strani vsebuje smernice za razvijalce: predstavi vse funkcionalnosti in koncepte, API dokumentacijo, opis in razlago sporočil o napakah ter prikaz izdelave manjše spletne aplikacije. Ponuja tudi povezavo na svoj YouTube kanal, ki vsebuje posnetke za različne teme in seminarje ter povezavo na tečaj, ki je dostopen na codeschool.com.

Ember dokumentacija vsebuje prikaz in predstavitev konceptov ogrodja, predstavitev in opis API ter krajši tečaj, ki prikaže izdelavo manjše aplikacije. Ponuja tudi svoj forum in IRC kanal, ki omogoča razvijalcem, da zastavijo vprašanja na temo ogrodja.

Backbone dokumentacija vsebuje pregled najpomembnejših konceptov in me-

tod ter krajših prikazov uporabe le-teh.

Velikokrat se pripeti, da samo dokumentacija na spletni strani ogrodja ni dovolj. Veliko lažje dobimo dodatne informacije in pomoč, če je ogrodje razširjeno med uporabniki. V tabeli 5.3 so prikazani podatki o virih, ki so bili zbrani 10. marca 2015.

Viri	**AngularJS**	**Ember.js**	**Backbone.js**
Zunanji moduli	1273-ngmodules	709-emberaddons	254-backplug
YouTube zadetki	102.000	9.660	15.900
StackOverflow vprašanja	82.049	14.276	17.545

Tabela 5.3: Prikaz zunanjih virov ogrodji.

Kot lahko vidimo, največ brezplačnih virov in odgovorov na vprašanja najdemo na temo AngularJS ogrodja. Seveda pa obstajajo tudi plačljivi viri v obliki knjig ter seminarjev na straneh, kot so npr. Pluralsight in Lynda. Stanje glede virov se lahko spremeni, saj razvijalci AngularJS načrtujejo novo verzijo 2.0, ki naj bi vsebovala veliko sprememb.

5.4 Testiranje

Testiranje je ena izmed najpomebnejših aktivnosti v razvojnem ciklu aplikacije. Omogoča, da izoliramo in odstranimo čimvečje število hroščev v naši aplikaciji. Prvi korak je pisanje testa, ki pade, nato pisanje kode in ponovno testiranje, dokler niso vsi testi pozitivni. Sprotno testiranje nam pokaže, ali naša aplikacija deluje tudi po tem, ko vpeljemo novo funkcionalnost.

AngularJS podpira Unit in E2E (End-to-End) testiranje. Pri Unit testiranju izoliramo del funkcionalnosti aplikacije in jo testiramo neodvisno od ostalih delov aplikacije. Pri E2E testiranju se postavimo v zorni kot uporabnika, ki ne ve vseh podrobnosti sistema, in testiramo večje enote aplikacije. Karma je orodje za testiranje, ki omogoča oba tipa testov. Za Unit teste podpira več različnih ogrodji kot, so Jasmine, Mocha in QUnit. Za ustvarjanje navideznih komponent pa uporabimo angular-mocks knjižnico. Za E2E testiranje lahko uporabimo orodji, kot sta PhantomJS ali CasperJS, ter ogrodje, kot je Protractor [4].

Spodnji primer prikazuje Jasmine unit test:

```
describe('Unit test', function(){
        it('test bo uspesen', function(){
                expect(true).toBe(true);
        });
});
```

Pri testiranju Backbone aplikacij imamo širok spekter možnosti. To so Jasmine, Mocha, QUnit kot ogrodja za testiranje. Spodnji primer nam prikazuje test v QUnit:

```
test('test()', function(){
   ok(test(), 'Opis rezultata');
})
```

Tudi Ember aplikacije lahko testiramo z več različnimi ogrodji za testiranje, ki so na voljo preko zunanjih adapterjev. Ponuja pomočnike za oba tipa testov. Za integracijske teste ponuja asinhrone in sinhrone pomočnike. Za Unit teste pa ponuja Ember-QUnit knjižnico.

5.5 Učinkovitost

Na učinkovitost aplikacije vpliva veliko dejavnikov: implementacijske odločitve razvijalcev, velikost ogrodja in vsa dodatna orodja, ki so potrebna za razvoj. Velikost ogrodja vpliva na to, kako hitro se bo stran naložila. V spodnji tabeli lahko vidimo prikaz velikosti posameznih ogrodji skupaj z njihovimi odvisnostmi in dodatnimi funkcionalnostmi.

AngularJS za razliko od ostalih dveh ogrodji nima nobenih odvisnosti, saj vsebuje svojo verzijo jQuery in sistem predlog, medtem ko za komunikacijo s strežnikom in implementacijo usmerjevalnika potrebuje dodatna modula, ki sta dostopna na domači spletni strani. Tabela 5.4 prikazuje velikost posameznega ogrodja, njihovih odvisnosti in poljubne dodatne funkcionalnosti.

Ogrodje	**Osnova**	**Odvisnosti**	**Dodatne funkcionalnosti**	**Skupaj**
AngularJS	44.9	//	1.8 (angular-resource), 2.1 (angular-route)	48.8
Ember	373	33.6 (jQuery), 121 (Handlebars)	387 (Ember Data)	914
Backbone	6.5	5.7 (Underscore), 33.6 (jQuery), 17kb (json2)	9.7 (Marionette), 4.2 (Epoxy)	76.7

Tabela 5.4: Prikaz velikosti ogrodji v KB.

Ember ima dve odvisnosti, in sicer sistem predlog v Handlebars in jQuery. Ember Data predstavlja nadgradnjo osnovnega sistema za upravljanje s podatki v aplikaciji in komunikacijo s strežnikom.

Backbone ima daleč najmanjšo osnovo, vendar za doseganje približno iste funkcionalnosti kot ostali dve ogrodji potrebuje veliko dodatnih orodij.

Poleg velikosti ogrodja na učinkovitost vplivajo še drugi dejavniki, kot so hitrost zagona; odzivnost na dogodke, sprožene s strani uporabnika; hitrost manipulacije DOM s strani ogrodja; upravljanje z viri itd.

Ena izmed takih primerjav je dostopna na Vue.js [22], ki prikaže implementacijo TodoMVC aplikacije in hitrost izvedbe akcij (dodajanje 100 elementov, označevanje opravljenih elementov, brisanje vseh elementov) za vsako ogrodje posebej. Testi so bili opravljeni v Firefox 36.0. Rezultati so vidni v tabeli 5.5, ki prikazuje skupni čas, ki ga je posamezno ogrodje potrebovalo za izvršitev vseh treh nalog.

Ogrodje	**Povprečje po 10 ponovitvah**
AngularJS	2088
Ember	2387
Backbone	1162

Tabela 5.5: Prikaz povprečnih časov v ms.

V tem primeru je zmagovalec Backbone. Vendar pa je to samo ena plat zgodbe, saj je razvijalec sam odgovoren za veliko stvari v Backbone aplikaciji, npr. DOM manipulacija.

Tabela 5.6 prikazuje čas posamezne akcije za eno ponovitev.

Akcija	**Backbone**	**Ember**	**AngularJS**
Dodajanje	655	1981	952
Označevanje	593	983	421
Brisanje	359	1279	343

Tabela 5.6: Prikaz meritev za posamezne akcije v ms.

Ember je počasnješi pri sestavljanju seznamov, saj vsakemu elementu doda opazovalca, ki spremlja spremembe vrednosti elementa. Na drugi strani je AngularJS hitrejši pri procesu sestavljanja seznamov, vendar pa mora pri vsaki spremembi preveriti vse elemente na scope. Če je model velik in kompleksen, pride do upočasnitve, saj pri tem opravlja tako imenovan "dirty checking". Tudi pri Backbone se lahko pojavijo težave z učinkovitostjo pri izrisu dolgih zbirk, saj lahko vsaka posodobitev na zbirki ponovno izriše vsak model v njej, kar pomeni, da moramo ponovno izračunati velikost in lokacijo za vsak element posebej v DOM drevesu.

Kot lahko vidimo, ima vsako ogrodje svoje dobre in slabe lastnosti glede na učinkovitost. Izbor ogrodja je odvisna tudi od tipa aplikacije, ki jo želimo razviti, in časa, ki ga imamo na razpolago.

5.6 Diskusija

Tabela 5.7 prikazuje pregled vseh funkcionalnosti, ki smo jih opisali.

Funkcionalnost	**AngularJS**	**Ember**	**Backbone**
Sistem predlog	Lastni sistem predlog.	Handlebars	Handlebars, Underscore

Usmerjevalnik	Ponuja lastno implementacijo, lahko tudi zunanjo.	Ponuja samo lastno implementacijo.	Ponuja lastno implementacijo, lahko tudi zunanjo.
Komunikacija s strežnikom	Ponuja $http in $resource modula za komunikacijo.	Uporablja jQuery AJAX metode ali Ember Data.	Omogoča RESTful operacije na modelu in zbirki.
Podatkovna povezava	Dvosmerna podatkovna povezava	Dvosmerna podatkovna povezava	Enosmerna podatkovna povezava

Tabela 5.7: Pregled funkcionalnosti.

Pri vseh ogrodjih smo opisali in primerjali glavne funkcionalnosti, ki nam omogočajo izdelavo SPA aplikacij, in ugotovili, kako se razlikujejo pri implementaciji le-teh. Glavne razlike se pri ogrodjih pokažejo pri sistemu predlog, saj AngularJS ponuja svojo implementacijo sistema predlog, medtem ko se ostali dve ogrodji zanašata na zunanje rešitve. Ember ponuja najboljšo implementacijo usmerjevalnika, saj omogoča definicijo vgnezdenih stanj, ko lahko to funkcionalnost pri ostalih dveh ogrodjih dobimo samo preko zunanjih rešitev. Razlike se pokažejo tudi pri podatkovni povezavi, saj ogrodji AngularJS in Ember.js ponujata dvosmerno podatkovno povezavo, Backbone.js pa je ne.

Vsa ogrodja smo razdelili od prvega do tretjega mesta glede na kriterije, kot so prilagodljivost, dokumentacijo in skupnost, testiranje in učinkovitost. Tabela 5.8 prikazuje razvrstitve ogrodij.

Ogrodje	**AngularJS**	**Ember**	**Backbone**
Prilagodljivost	2.	3.	1.
Dokumentacija in skupnost	1.	3.	2.
Testiranje	1.	2.	3.
Učinkovitost	2.	3.	1.

Tabela 5.8: Pregled primerjanih kriterijev.

Za najbolj prilagodljivega se izkaže Backbone, saj prepušča razvijalcu odločitve o strukturi aplikacije in implementaciji določenih funkcionalnosti. Vsa ogrodja ponujajo spletne strani z opisi konceptov in funkcionalnosti, vendar pa ima AngularJS največjo skupnost, v kateri najdemo vire za pomoč in učenje. Ember ponuja podobne funkcionalnosti kot AngularJS, vendar zaradi pogostega spreminjanja API postane dokumentacija zastarela in primeri razvoja aplikacij nedelujoči. S tem pa razvijalcu otežuje učni proces, ker ima težave pri iskanju virov za pomoč in delujočih prikazov delovanja posameznih komponent ogrodja. Struktura AngularJS aplikacij omogoča najboljšo osnovo za učinkovito testiranje. Backbone omogoča direktno manipulacijo DOM v Pogledu, s tem pa otežuje izolacijo določenih komponent in njihovo testiranje. Pri analizi učinkovitosti se najbolj izkaže Backbone, saj je najhitrejši pri operacijah dodajanja, označevanja in brisanja elementov iz seznama ter ima najmanjšo velikost, kar pomeni krajši nalagalni čas.

Veliko prednost AngularJS predstavlja obsežen nabor virov in literature, s katero si lahko pomagamo, ko zaidemo v težave. AngularJS daje velik poudarek na testiranju aplikacije. Omogoča ustvarjanje komponent za večkratno uporabo in delitev aplikacije na logične dele z jasno začrtanimi nalogami. Zato bi na tem mestu najbolj priporočil AngularJS, ki prav zaradi obsežne skupnosti in dokumentacije ter pravim razmerjem med fleksibilnostjo in restriktivnostjo pri posameznih konceptih, nudi najbolj uravnoteženo rešitev za razvijalce.

Poglavje 6

Razvoj aplikacije

6.1 Uvod

Namen razvoja aplikacije je na praktičnem primeru prikazati izgradnjo SPA aplikacije z uporabo MVC ogrodja ter uporabo komponent, kot so moduli, direktive, usmerjevalnik, krmilnik in sodelovanje med njimi.

6.2 Izbor ogrodja

Za razvoj testne aplikacije je bilo izbrano ogrodje Angular. Ima veliko skupnost, kar posledično pomeni veliko različnih virov informacij, mnenj in nasvetov pri spoznavanju in učenju tega JavaScript ogrodja. Poudarja modularni razvoj ter ponuja funkcionalnosti, kot so filtri, direktive, validacija, s tem pa pospeši razvoj aplikacij.

6.3 Priprava HTTP/RESTful storitve

Spletno storitev, ki bo predstavljala vir informacij naše aplikacije, bomo razvili s pomočjo ASP.NET Web API in Entity Framework 6, ki bo predstavljal našo objektno—relacijsko preslikovalno ogrodje. V podatkovni bazi bomo hranili tudi geografske podatke, ki jih bomo uporabili za prikaz objektov na zemljevidu. To nam SQL Server omogoča preko Spatial tipov, nas pa še posebej zanima tip Ge-

ography, ki nam omogoča hranjene podatkovnih tipov poligon in multipoligon. Microsoft.SqlServer.Types paket nam omogoča uporabo tipov DbGeography in DbGeometry v povezavi z Entity Framework ogrodjem. Na spodnjem primeru vidimo dodajanje potrebne vrstice v Global.asax.cs, ki nam omogoča uporabo Spatial tipov v naši aplikaciji.

```
protected void Application_Start(){
   //ostale nastavitve
   SqlServerTypes.Utilities.LoadNativeAssemblies(Server.MapPath("~/bin"));
}
```

V naši aplikaciji potrebuje zemljevid za predstavitev objektov podatke v GeoJson formatu, zato namestimo še GeoJSON.NET NuGet paket, ki vsebuje pretvornike za serializacijo in deserializacijo podatkov GeoJson. Na spodnjem primeru lahko vidimo prikaz predstavitve geometrijskih podatkov v GeoJson formatu.

```
{ "type": "FeatureCollection",
    "features": [
      { "type": "Feature",
        "geometry": {"type": "Point", "coordinates": [102.0, 0.5]},
        "properties": {"prop0": "value0"}
        },
      { "type": "Feature",
        "geometry": {
          "type": "LineString",
          "coordinates": [
            [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]
            ]
          },
        "properties": {
          "prop0": "value0",
          "prop1": 0.0
          }
        },
      { "type": "Feature",
```

```
    "geometry": {
      "type": "Polygon",
      "coordinates": [
        [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
          [100.0, 1.0], [100.0, 0.0] ]
        ]
    },
    "properties": {
      "prop0": "value0",
      "prop1": {"this": "that"}
      }
    }
  ]
}
```

Omogočili bomo še CORS (angl. Cross Origin Resource Sharing), ki je W3C standard, ki dovoljuje strežniku določene zahteve različnega porekla in zavrne druge. CORS je varnejši kot zgodnejše tehnike, kot je JSONP [20]. CORS omogočimo z namestitvijo NuGet paketa Microsoft.AspNet.WebApi.Cors ter dodajanjem config.EnableCors() v WebApiConfig.Register metodo in [EnableCors] atributa na naš krmilnik, kot je vidno na spodnjem primeru.

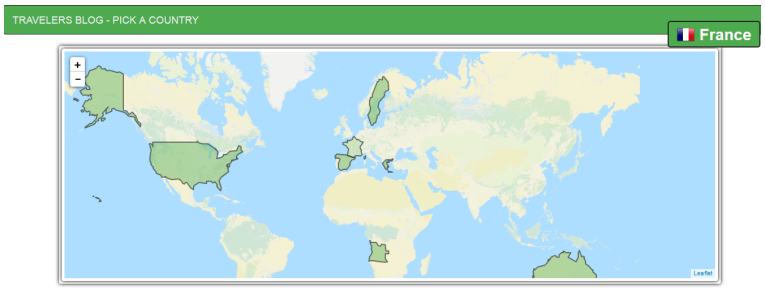
```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {

        config.EnableCors();

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
```

```
        }
    }
}
```

```
[EnableCors(origins: "http://example.com/foo.html", headers: "*",
    methods: "*")]
 public class TestController : ApiController
```

6.4 Prikaz aplikacije

Aplikacija predstavlja potovalni blog, kjer je uporabniku omogočena navigacija po zemljevidu in izbira posamezno označene države na zemljevidu, ki ga vodi do seznama blog zapisov za posamezno državo. Za prikaz zemljevida uporabimo An-



Slika 6.1: Prikaz začetne strani.

gular direktivo angular-leaflet-directive. Ta nam omogoča nastavljanje parametrov zemljevida in registracijo poslušalcev v krmilniku. Spodnji primer prikazuje definicijo direktive v HTML predlogi.

```
    <leaflet id="zemljevid"
            center="vm.center"
            events="vm.events"
            maxbounds="vm.maxbounds"
            tiles="vm.tiles"
            defaults="vm.defaults"
```

```
            geojson="vm.geojson">
    </leaflet>
```

Če želimo delujočo direktivo, jo moramo registrirati v glavnem modulu aplikacije in vključiti naslednje datoteke: leaflet.js, leaflet.css (dostopno na: `http://leafletjs.com/`) ter angular-leaflet-directive.js (dostopno na: `http://ngmodules.org/modules/angular-leaflet-directive`). Spodnji primer prikazuje registracijo v glavnem modulu aplikacije.

```
angular.module('app', [
    // Angular modules
    'ui.router',
    'ngResource',
    'ngRoute',
    'ngAnimate',
    // Custom modules
    'common.services',
    'directives',
    // 3rd Party Modules
    'leaflet-directive'
]);
```

Poslušalce definiramo v krmilniku na naslednji način:

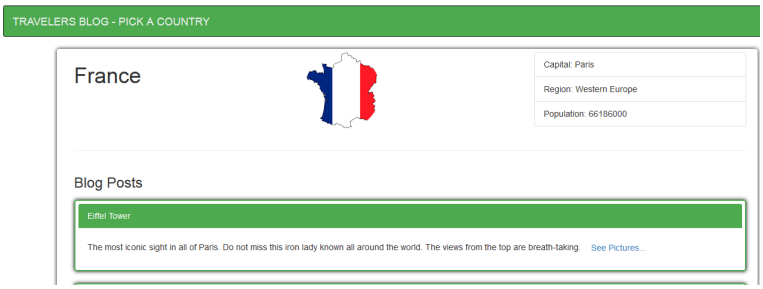
```
$scope.$on("leafletDirectiveGeoJson.zemljevid.mouseover",
    function (ev, leafletPayload) {
    countryMouseover(leafletPayload.leafletObject.feature,
        leafletPayload.leafletEvent);
});

$scope.$on("leafletDirectiveGeoJson.zemljevid.click",
    function (ev, leafletPayload) {
    countryClick(leafletPayload.leafletObject,
        leafletPayload.leafletEvent);
});
```

```
    $scope.$on("leafletDirectiveGeoJson.zemljevid.mouseout",
        function (ev, leafletPayload) {
        countryMouseout(leafletPayload.leafletObject.feature,
            leafletPayload.leafletEvent);
    });
```

Za navigacijo v aplikaciji uporabimo AngularUI Router, ta nam omogoča definicijo stanj in vgnezdenih, poimenovanih ter paralelnih pogledov. Router definiramo na naslednji način:

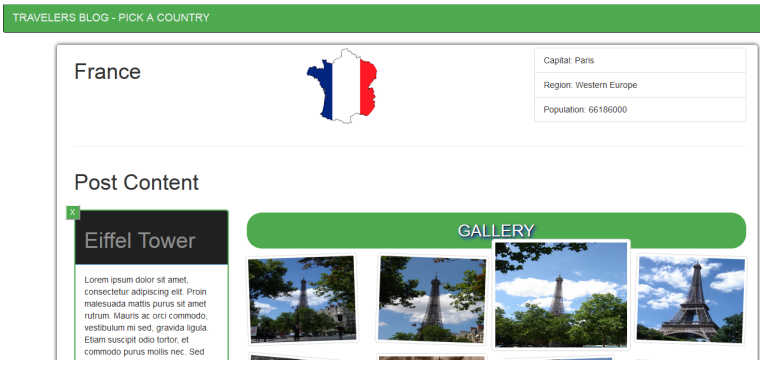
```
.config(function ($stateProvider, $urlRouterProvider) {

    $urlRouterProvider.otherwise('/map');
    $stateProvider
        .state('map', {
            url: '/map',
            templateUrl: 'app/map/map.html',
            controller: 'mapCtrl as vm'
        })
        .state('country', {
            abstract: true,
            url: '/country/:id',
            templateUrl: 'app/country/country.html',
            controller: 'countryCtrl as vm'
        }).state('country.posts', {
            url: '/posts',
            templateUrl: 'app/country/countryPosts.html'
        }).state('country.post', {
            url: '/post/:postId',
            templateUrl: 'app/post/postDetail.html',
            controller: 'postCtrl as vm'
        });
```

Stanje country je abstraktno, kar pomeni, da ga ne moremo direktno aktivirati, vendar pa lahko dostopamo do njegovih gnezdenih stanj. Slika 6.2 prikazuje aplikacijo, ko je aktivno stanje posts.



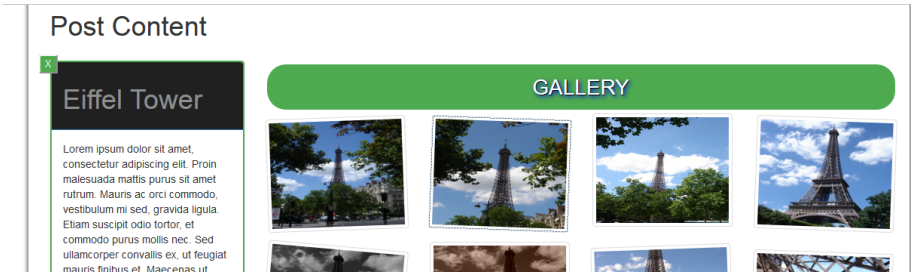
Slika 6.2: Gnezdeni pogled posts.

Slika 6.3 prikazuje aplikacijo, ko je aktivno stanje post.

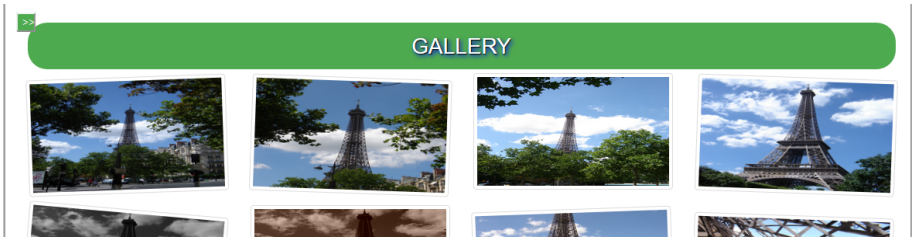


Slika 6.3: Gnezdeni pogled post.

Animacije na elementih so mogoče z ngAnimate modulom. Z uporabo tega modula lahko dodajamo ali odstranimo CSS razrede, ko se zgodijo določeni dogodki na elementu. Elementi, ki jim lahko določimo animacije, so ngRepeat, ngSwitch, ngView. CSS razrede določimo v animations.css. Na sliki 6.4 lahko vidimo prikaz stanja pred animacijo, na sliki 6.5 pa stanje po izvršeni animaciji.

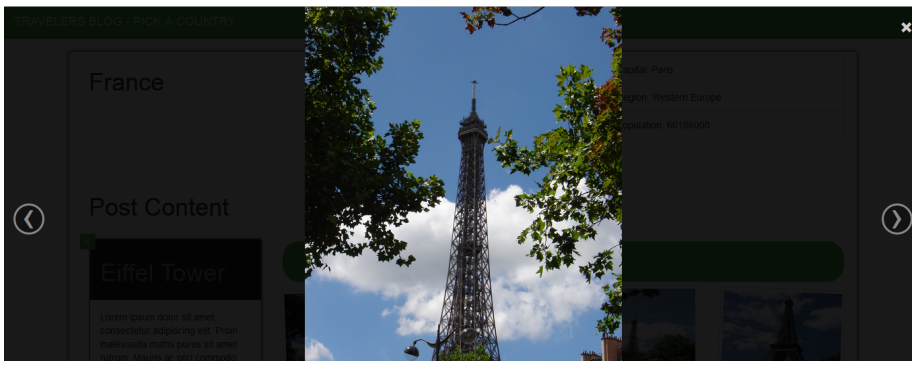


Slika 6.4: Prikaz pred aktiviranjem animacije.



Slika 6.5: Prikaz po aktivaciji animacije.

Slika 6.6 prikazuje aktivacijo galerijo, ki jo sprožimo s klikom na posamezno sliko v seznamu.



Slika 6.6: Prikaz aktivacije galerije.

Poglavje 7

Sklepne ugotovitve

V diplomskem delu smo primerjali tri JavaScript ogrodja: AngularJS, Ember.js in Backbone.js. Opisali smo njihovo implementacijo arhitekturnega modela MVC in gradnike, ki predstavljajo posamezne komponente tega modela.

Pri primerjavi funkcionalnosti se ogrodja najbolj razlikujejo pri podpori dvosmerne podatkovne povezave. Odstopanja se pokažejo tudi pri implementaciji usmerjevalnika, saj Ember ponuja funkcionalnosti, ki ju ostali dve ogrodji ne. Poleg funkcionalnosti smo primerjali tudi druge zelo pomembne vidike, kot so prilagodljivost, dokumentacija in skupnost, testiranje ter učinkovitost posameznega ogrodja. Za najbolj prilagodljivega se izkaže Backbone.js, medtem ko sta ostali dve ogrodji bolj restriktivni. Največ brezplačnih virov in odgovorov najdemo na temo AngularJS ogrodja.

Za razvoj spletne aplikacije smo uporabili ogrodje AngularJS, ki ponuja veliko uporabnih funkcionalnosti, ki pospešijo razvoj. Veliko prednost predstavlja obsežen nabor virov in literature, s katero si lahko pomagamo, ko zaidemo v težave. Aplikacija predstavlja potovalni blog, kjer lahko uporabniki z navigacijo po zemljevidu izberejo označene države, ki jih je avtor bloga obiskal. Z izborom posamezne države uporabniki dobijo dostop do blogov za posamezno državo. Posamezni blogi prikažejo vsebino bloga in galerijo slik, ki jih lahko uporabnik pregleduje.

Kljub temu da Ember ponuja vse funkcionalnosti, ki jih ponuja AngularJS, bi na tem mestu najbolj priporočil AngularJS zaradi obsežne skupnosti in dokumentacije, ki je velika pomankljivost Ember. Nudi tudi pravo razmerje med fleksibilnostjo in restriktivnostjo pri posameznih konceptih in nudi najbolj urav-

noteženo rešitev za razvijalce. Backbone ponuja najmanjši nabor funkcionalnosti, predvsem pa ne ponuja funkcionalnosti, ki so postale standard pri izdelavi SPA aplikacij. S tem pa podaljšuje čas razvoja, zato bi to ogrodje odsvetoval.

Seveda pa ta tri ogrodja ne predstavljajo vseh možnosti, ki jih ima razvijalec pri izbiri JavaScript MVC ogrodja. Možnosti je veliko in pojavljajo se vedno nove rešitve na tem področju, kot so Knockout.js, Batman.js, Meteor, CanJS itd.

Primerjava ogrodij služi kot dober vir informacij za razvijalce, ki vstopajo v razvoj SPA aplikacij. Nudi vpogled v ogrodja in njihovo delovanje, s tem pa olajša izbiro.

Literatura

[1] A. Freeman, Pro AngulaJS, str. 330–584

[2] A. Lerner, Ng-book - The Complete Book on AngularJS, str. 8

[3] A. Lerner, Ng-book - The Complete Book on AngularJS, str. 132–136

[4] A. Lerner, Ng-book - The Complete Book on AngularJS, str. 295–315

[5] A. Osmani, Developing Backbone.js Applications, str. 2–61

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture Volume 1:A System of Patterns, str. 126–128

[7] J. Cravens, T. Brady, Building Web Apps with Ember.js, str. 8–103

[8] J. Sugrue, Beginning Backbone.js Applications, str. 6–70

[9] M. S. Mikowski, J. C. Powell, Single Page Web Applications – JavaScript end-to-end , str. 1–8

[10] N. C. Zakas, Professional JavaScript for Web Developers, 3rd Edition, str. 1–10

[11] N. C. Zakas, J. McPeak, J. Fawcett, Wrox Professional Ajax, 2nd Edition, str. 25–28

[12] R. Branas, AngularJS Essentials, str. 8–18

[13] V. Mirgorod, Backbone.js Cookbook, str. 72

Spletni viri:

[14] AngularJS: Developer Guide, 11. februar 2015, dostopno na: https://docs.angularjs.org/guide

[15] A. Osmani, , Developing Backbone.js Applications, 12. februar 2015, dostopno na: http://addyosmani.github.io/backbone-fundamentals/

[16] Ember.js-Concepts:Core Concepts, 12. februar 2015, dostopno na: http://emberjs.com/guides/concepts/core-concepts/

[17] Ember.js-Components: Defining a Component , 12. februar 2015, dostopno na: http://emberjs.com/guides/components/defining-a-component/

[18] Ember.js-Models: Introduction, 12. februar 2015, dostopno na: http://emberjs.com/guides/models/

[19] Ember.js-Templates: Input Helpers, 13. februar 2015, dostopno na: http://emberjs.com/guides/templates/input-helpers/

[20] Enabling Cross-Origin Requests in ASP.NET Web API 2 — The ASP.NET Site, 28. januar 2016, dostopno na: http://www.asp.net/web-api/overview/security/enabling-cross-origin-requests-in-web-api#enable-cors

[21] Historical trends in the usage of client-side programming languages, 10. januar 2015, dostopno na: http://w3techs.com/technologies/history_overview/client_side_language/all

[22] Performance Comparisons, 2. marec 2015, dostopno na: http://legacy.vuejs.org/perf/

[23] The Model-View-ViewModel (MVVM) design pattern for WPF, 9. februar 2015, dostopno na: https://msdn.microsoft.com/en-us/magazine/dd419663.aspx