Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Erik Ušaj

Skladi tehnologij in ogrodja za celostni odjemalsko-strežniški razvoj aplikacij

diplomsko delo

visokošolski strokovni študij računalništva in informatike

Mentor: viš. pred. dr. Borut Batagelj

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V svojem delu predstavite aktualne sklade tehnologij in ogrodja, ki temeljijo na programskem jeziku JavaScript. Predstavite, kako lahko s pomočjo predstavljenih tehnologij razvijemo mobilne in namizne aplikacije, ter ali lahko uporabimo sklade tehnologij tudi za razvoj spletnih odjemalcev in strežniških sistemov. Predstavljene tehnologije uporabite v praktičnem primeru aplikacije s področja računalniškega vida.

Zahvala mentorju, delodajalcem, sodelavcem, prijateljem in vsem, ki ste me spremljali in spodbujali ob študiju.

Diplomsko delo posvečam vsem, ki (še vedno) ne zaupajo programskemu jeziku JavaScript in vsem delodajalcem na področju informacijske tehnologije, ki premalo vlagajo v izobraževanje, prenos znanja in nego kulture, ki spodbuja strokovno rast zaposlenih.

My thesis is dedicated to anyone, who (still) doesn't trust JavaScript and to all IT employers not investing enough in education, knowledge transfer and fostering of corporate culture that encourages further professional growth of employees.

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 ECMAScript (JavaScript) 1

1.2 Vstopne točke v sklade tehnologij 5

1.3 Metodologija ocenjevanja ustreznosti 10

2 Sklad tehnologij MEAN **13**

2.1 MongoDB 15

2.2 Express.js 19

2.3 Angular.js 21

2.4 Node.js 24

3 Ogrodje Meteor **29**

3.1 Primerjava MEAN in Meteor 30

3.2 Struktura map aplikacije Meteor 34

3.3 Protokol DDP 36

3.4 Protokol HTTP 36

3.5 Procesor predlog Blaze 37

3.6 Podpora Angular.js 38

3.7 Podpora Angular 2 39

3.8 Podpora React 40

3.9 Apache Cordova 42
3.10 Podatkovni sklad Apollo 44
3.11 Mantra 47

4 Dodatne tehnologije za celostni razvoj 53
4.1 Tehnologije za namizne aplikacije 53
4.2 Tehnologije za storitve v oblaku 57
4.3 Razvoj aplikacij za internet stvari 59

5 Izbira sklada tehnologij za izbran praktični primer 63
5.1 Zajem slike na napravah 63
5.2 Obdelava in analiza zajete slike 67
5.3 Nalaganje slik na spletni strežnik 68

6 Razvoj aplikacije 71
6.1 Priprava razvojnega okolja 71

7 Sklepne ugotovitve 77

Literatura 81

Seznam uporabljenih kratic

kratica	angleško	slovensko
AJAX	Asynchronous JavaScript and XML	Asinhroni JavaScript in XML
API	Application Programming Interface	aplikacijski programski vmesnik
AMQP	Advanced Message Queuing Protocol	Odprtokodni protokol za sporočanje
AWS	Amazon Web Services	spletne storitve Amazon
BSON	Binary JavaScript Object Notation	binarna objektna notacija JavaScript
CEF	Chronium Embedded Framework	vgradno ogrodje Chronium
CLI	Command Line Interface	vmesnik ukazne vrstice
CMS	Content Managment System	Sistem za upravljanje vsebin
CPU, CPE	Central Processing Unit	centralna procesna enota
CRM	Customer Relationship Management	Sistem upravljanja odnosov s strankami

CRUD	Create, Read, Update and Delete	Ustvarjanje, branje, posodabljanje in brisanje
CSS	Cascading Style Sheet	kaskadne slogovne predloge
CV	Computer Vision	računalniški vid
DBA	Database Administrator	Skrbnik podatkovnih baz
DBMS	Database Management System	sistem za upravljanje podatkovnih baz
DDP	Distributed Data Protocol	protokol za poizvedbe, posodobitve in sinhronizacijo v realnem času
DOM	Document Object Model	objetkni model dokumenta
EC2	Elastic Compute Cloud (Amazon)	prilagodljiva infrastruktura v oblaku
ECMA	European Computer Manufacturers Association	Ecma International
EN	European Standard	evropski standard
EJS	Embedded JavaScript templates	sistem predlog EJS
ES	scripting-language specification standardized by Ecma International	standardizirana specifikacija skriptnega jezika organizacije Ecma International
ES5	ECMA-262 5th Edition	5. izdaja ECMA-262
ES6	ECMA-262 6th Edition	6. izdaja ECMA-262

GUI	Graphical User Interface	grafični uporabniški vmesnik
HTML	Hyper Text Markup Language	jezik za označevanje nadbesedila
HTTP	HyperText Transfer Protocol	protokol za prenos nadbesedila
I/O	Input/Output	Vhod/Izhod
IDC	International Data Corporation	ponudnik tržnih informacije iz IT
IDE	Interactive Development Environment	interaktivno razvojno okolje
IoT	Internet of Things	internet stvari
ISO	International Organization for Standardization	mednarodna organizacija za standardizacijo
IT	Information Technology	informacijske tehnologije
JIT	Just-In-Time	Sprotno prevajanje
JS	JavaScript	JavaScript
JSX	XML-like syntax extension to ECMAScript	razširitev ES z elemetni podobnimi XML
JSON	JavaScript Object Notation	objektna notacija JavaScript
LAMP	Linux, Apache, MySQL, PHP	sklad tehnologij LAMP
MDG	Meteor Development Group	razvojna skupina Meteor

MEAN	Mongo DB, Express, Angular, Node	sklad tehnologij MEAN
MVC	Model–View–Controller	arhitektura, ki temelji na ločenih nivojih
MVP	Model-View-Presenter	arhitektura, ki temelji na ločenih nivojih
MVVM	Model–View–View-Model	arhitektura, ki omogoča ločevanje razvoja in obikovanja GUI
MVW	Model-View-Whatever	arhitektura MV-karkoli
NoSQL	Not only SQL	oznaka nove generacije podatkovnih baz, ki ne uporabljajo (samo) SQL
NPM	Node Package Manager	upravitelj pakeotv Node
OAI	Open API Initiative	iniciativa za odprte APIje
OCR	Optical Character Recognition	optično prepoznavanje znakov
PHP	PHP: Hypertext Preprocessor	predprocesor nadbesedila
PoC	Proof of Concept	preskus koncepta
POSIX	Portable Operating System Interface	nabor standardov za združljivost operacijskih sistemov
PSD	(Adobe) Photoshop Data file	podatkovna datoteka Photoshop

RAM	Random Access Memory	spomin z direktnim dostopom, delovni pomnilnik
RDBMS	Relational Database Management System	relacijski sistem za upravljanje podatkovnih baz
REST	Representational State Transfer	Predstavitveni prenos stanja
ROM	Read-Only Memory	bralni pomnilnik
RPC	Remote Procedure Call	klic oddaljene procedure
RTC	Real Time Communication	komunikacije v realnem času
SDK	Software Development Kit	programska orodja za razvoj
SIST	Slovenian Institute for Standardization	Slovenski inštitut za standardizacijo
SOAP	Simple Object Access Protokol	protokol za razvoj spletnih servisov, ki temelji na XML
SQL	Structured Query Language	strukturirani povpraševalni jezik za delo s podatkovnimi bazami
STB	Set Top Box	Naprava za sprejem kabelske ali IP-TV
TLS	Transport Layer Security	kriptografski protokol
UI	User Interface	uporabniški vmesnik

UID	User Interface Design	obikovanje interaktivnosti uporabniškega vmesnika ob sodelovanju z razvijalcem
URI	Uniform Resource Identifier	enoličen idetifikator vira
URL	Uniform Resource Locator	enoličen naslov spletne strani
UXD, UED	User Experience Design	proces raziskav in izboljšav uporabnosti izdelka in interakcije uporabnik-izdelek
VM	Virtual Machine	virtualni stroj
XHTML	Extensible Hypertext Markup Language	razširitev jezika za označevanje nadbesedila
XML	Extensible Markup Language	format potatkov za izmenjavo strukturiranih dokumentov v spletu
XSLT	XML Stylesheet Language for Transformations	XML predloga za preoblikovanje
XSS	Cross-Site Scripting	napad izveden s pomočjo vrivanja skript

Povzetek

Naslov: Skladi tehnologij in ogrodja za celosten odjemalsko-strežniški razvoj aplikacij

S tem delom želimo analizirati aktualne sklade tehnologij (*angl. technology stacks*) in ogrodja JavaScript (JS) za celosten odjemalsko-strežniški razvoj aplikacij (*angl. fullstack development frameworks*): od spletnih odjemalcev, mobilnih in namiznih aplikacij (*angl. desktop applications*) do strežniških zalednih sistemov in povezljivosti s spletnimi storitvami v oblaku. Fokus analize bomo posvetili skladu tehnologij *MEAN* (*MongoDB*, *Express.js*, *Angular.js*, *Node.js*) in ogrodju *Meteor*, ki dodatno rešuje gradnjo mobilnih aplikacij s pomočjo odprtokodnega ogrodja *Apache Cordova*. Ogledali si bomo orodja za razvoj aplikacij v JS za namizne operacijske sisteme (*NW.js*, *Electron*) ter možnost uporabe JS v scenarijih interneta stvari (*angl. internet of things, IoT*) (npr. povezane naprave *Raspberry Pi*). Kot praktični problem smo si zastavili zajem slike na različnih napravah, obdelavo zajete slike na odjemalcu ali strežniškem sistemu oz. uporabo obstoječih spletnih servisov za računalniški vid oz. slikovno prepoznavanje. Ugotoviti želimo ali je možna vzpostavitev lastne tovrstne spletne storitve razvite v JS, kje se lahko pojavijo omejitve tehnologij JS in načine kako jih lahko zaobidemo.

Ključne besede: skladi tehnologij, ogrodja, celostni razvoj aplikacij, MEAN, Meteor, MongoDB, Express.js, Angular, Node.js, Electron, storitve v oblaku, IoT.

Abstract

Title: Technology stacks and frameworks for full-stack application development

This work aims providing a comprehensive overview and analysis of current JavaScript (JS) technology stacks and frameworks for full-stack application development: from web clients, mobile and desktop applications to server applications and cloud-connected services. Analysis shall focus on MEAN technology stack and frameworks such as Meteor which also tries to leverage mobile app development using Apache Cordova framework. We will include an overview of available JS build tools for desktop application development and take a look at use-cases for JS-based IoT development (i.e. connected Raspberry Pi devices). Image capture on different devices will be used as use-case scenario where image processing can be done either on client or server-side or using existing CV/AI services. We should evaluate if JS is suitable for developing CV services, establish its limitations and possible workarounds.

Keywords: technology stacks, frameworks, full-stack development, MEAN, Meteor, MongoDB, Express.js, Angular, Node.js, Electron, cloud services, IoT.

Poglavje 1

Uvod

Sklade tehnologij uporabljamo, ko nam ena sama tehnologija ne omogoča doseganja želenih rešitev, predstavniki skladov tehnologij so: AJAX, LAMP, MEAN, itd.
Ajax npr. ni tehnologija, je v bistvu sklop več tehnologij, vsaka ima svojo rast, ko jih uporabimo skupaj, nam prispevajo nove zmogljivosti [1]. Poglejmo, kaj vključuje AJAX, katerega ime je skovanka iz besed asinhroni, JavaScript in XML:

1. standardni predstavitveni nivo z uporabo XHTML in CSS,
2. dinamični prikaz s pomočjo DOM,
3. izmenjavo podatkov z uporabo XML in XSLT
4. XMLHttpRequest za asinhrono pridobivanje podatkov in
5. JavaScript, kot povezovalni element vsega.

1.1 ECMAScript (JavaScript)

Programski jezik *ECMAScript* ®, mnogim bolj poznan pod imenom *JavaScript* (JS), danes odlično pokriva tako odjemalski, kot strežniški del razvoja.

JS je visoko-nivojski, dinamičen, netipiziran interpretiran programski jezik, ki je zelo primeren za objektno-orientirano in funkcijsko programiranje [2].

Njegova uporabnost sega tudi na področje podatkovnih baz, ki ne uporabljajo (samo) jezika SQL (*NoSQL*), kjer se notacija *JSON* [3, 4, 5] lahko uporablja tako za poizvedbe, kot za podatkovne sheme. Njegov potencial vidijo tudi vsi večji proizvajalci programskih rešitev in ponudniki storitev v oblaku. Zato se bomo osredotočili na tehnologije, ki temeljijo na JS, umetnem jeziku, katerega tekoče znanje v današnjem času za razliko od znanja esperanta lahko omogoča lažjo zaposljivost [6].

1.1.1 Zgodovina

JavaScript

Jezik avtorja Brendana Eicha se je ob nastanku v maju 1995 imenoval *Mocha*, že v prvem letu obstoja je bil najprej preimenovan v *LiveScript* in še pred iztekom leta v *JavaScript* [7]. Prvotno je služil predvsem reševanju preprostih problemov v brskalnikih *Netscape* in kasneje *Mozilla*. Okrog jezika se je tekom let in procesov standardizacije (Slika 1.1) in boja za svoj prav med različnimi ponudniki brskalnikov razvilo lepo število dialektov med katerimi lahko najdemo tako nestandardne implementacije, kot tudi kako mrtvo vejo.

ECMAScript

Med letoma 1996 in 1997 je jezik prevzela *ECMA* z namenom standardizacije in implementacije standarda na brskalnikih različnih proizvajalcev programske opreme. Prva izdaja standarda nosi uradno ime *ECMA-262 Ed.1: ECMAScript*, ki definira ES kot:

Jezik za pisanje skript (*angl. scripting language*), ki se uporablja za manipulacijo, prilagajanje in avtomacijo objektov v že obstoječih sistemih [8].

Ter kot: Spletni skriptni jezik (*angl. web scripting language*), ki nudi mehanizme poživljanja spletnih strani brskalnikov in izvajanje strežniških operacij kot del spletne odjemalsko-strežniške arhitekture [8].

Vidimo lahko, da je standard že ob nastanku predvideval možnost uporabe ES v odjemalsko-strežniških rešitvah tako v brskalniku, kot tudi na strežniku z namenom distribuiranega izvajanja med odjemalcem in strežnikom z namenom omogočanja prilagojenega *uporabniškega vmesnika* (*angl. user interface, UI*) za spletne aplikacije.

Standardizacija se je nadaljevala v okviru cikličnih izdaj standarda. Tretja izdaja uvaja regularne izraze, boljše rokovanje nizov, nove kontrolne stavke, rokovanje izjem s konstruktom `try { } catch (e){ } finally { }` . Po tretji izdaji se je začela masovna uporaba z implementacijo v vseh brskalnikih.

Četrta izdaja je bila sicer rezervirana a nikoli uporabljena v procesu objav standardov ECMA, zato formalno ne obstaja [9].

ES5

Peta izdaja *ECMA-262 5th edition*, dodaja podporo novim funkcionalnostim: dostop lastnostim, refleksivno kreiranje in inšpekcijo objektov, kontrolo programa in atributov lastnosti, dodatne funkcije za manipulacijo polj, podporo za JSON, in strikten način (*angl. strict mode*) z izboljšanim preverjanjem napak in varnostjo. Od leta 2012 vsi moderni brskalniki [10] v celoti podpirajo različico ES 5.1 [11].

ES6

Šesta izdaja *ECMA-262 6th edition* [12] predstavlja naslednjo generacijo ES in prinaša številne nove funkcionalnosti.
ES6 ločuje med spremenljivkami `let` in konstantami `const` .
Pri nizih poleg dodatnih metod prinaša tudi oblikovanje nizov s pomočjo predlog `let imeInPriimek = '$(ime), $(priimek)'` .
Nove metode so na voljo tudi za polja, novost je tudi operator (`...`) za razširjanje polj (*angl. spread*) in možnost razstavljanja (*angl. destructuring*) objektov in polj `let [x, y] = [1, 2]; //x = 1, y = 2` .
Pri funkcijah je vpeljana možnost določanja privzetih vrednosti vhodnih vrednosti `function vsota(x = 0, y = 0){ return x + y; }` . ES6 uvaja tudi

definicije funkcij z zapisom s puščico `=>` .
Novost sta tudi rezervirani besedi `export` in `import` , ki omogočata uvažanje in izvažanje iz modulov.
ES6 vpeljuje sintakso za razrede. Nekateri verjamejo, da je to v nasprotju prototipni naravi JS, drugi so prepričani, da so razredi pomembni za lažji prehod iz drugih programskih jezikov. Kakor koli že, podpora razredom [13] je sintaktični priboljšek obstoječemu prototipnemu dedovanju. Razrede gradimo s pomočjo rezervirane besede `class` in metode `constructor()` (Koda 1.1).

```
class Vozilo {
   constructor(ime) {
      this.ime = ime;
      this.tip = 'vozilo';
   }
   vrniIme() {
      return this.ime;
   }
}

// Ustvarimo primerek
let mojeVozilo = new Vozilo('Skiro');

class Avto extends Vozilo {
   constructor(ime) {
      super(ime);
      this.tip = 'avto'
   }
}

let mojAvto = new Avto('Mazda');

mojAvto.vrniIme();           // Mazda
mojAvto instanceof Avto;     // true
mojAvto instanceof Vozilo;   // true
```

Koda 1.1: Primer sintakse razredov v ES6.

TypeScript

TypeScritp je odprtokodni programski jezik, ki si ga lahko predstavljamo kot nadmnožico JS, saj uporablja enako sintakso in semantiko [14]. Odprtokodni jezik, razvit pod okriljem družbe Microsoft, je plod razvoja Andersa Hejlsberga, ki je med drugim glavni arhitekt programskega jezika *C#* in ustvarjalec jezikov *Delphi* in *Turbo Pascal*. Izvorno kodo TypeScript lahko prevedemo (*angl. transcompile*) v izvorno kodo JS, ki jo lahko izvajamo v vsakem brskalniškem okolju in okolju Node.js, ki podpira implementacijo ES3 ali novejšo.

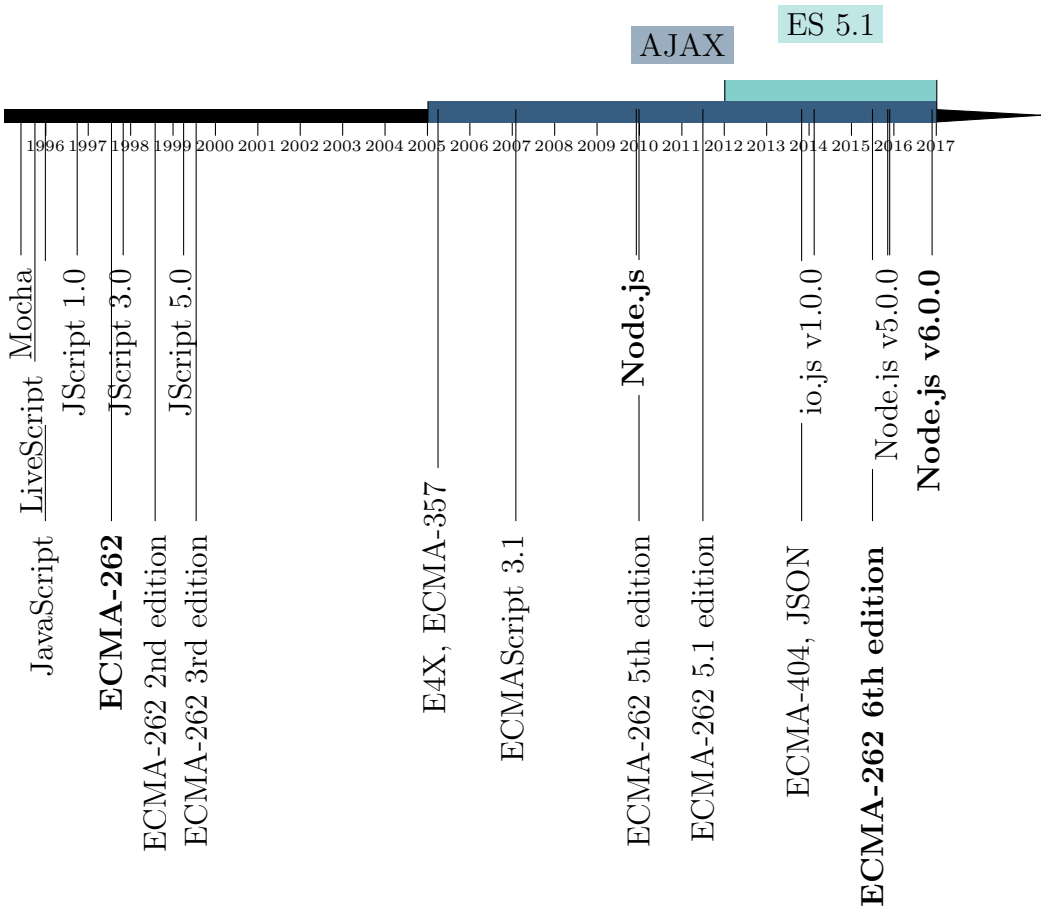
1.2 Vstopne točke v sklade tehnologij

Vstopna točka v sklad tehnologij je običajno ena izmed tehnologij, s katero se razvijalec sooči tekom analize zahtev in raziskave v okviru programskih izzivov bodisi za strežniške aplikacije ali odjemalske aplikacije. Za razvijalce, ki se ukvarjajo s podatkovnimi zbirkami ali izzivi učinkovitega hranjenja podatkov, bo tako naravna vstopna točka podatkovna zbirka *MongoDB* 2.1, za razvijalce spletnih strani mogoče *Express.js* 2.2 ali *Node.js* 2.4, za razvijalce uporabniških vmesnikov pa *Angular.js* 2.3. Neglede na vstopno točko, se uporabniki prej ali slej soočijo s težavami in začnejo spoznavati dodatne prednosti, ki jih prinaša uporaba celovitejšega sklada tehnologij.

Na primerih uporabe, ki izvirajo iz dejanskih izkušenj razvoja odjemalsko-strežniških aplikacij, si najprej oglejmo nekaj možnih vstopnih točk v sklad tehnologij *MEAN* oz. v ogrodje *Meteor*.

1.2.1 Orodje za vzpostavitev testnih primerov

Naše prvo spoznavanje s tehnologijami sklada *MEAN* se je začelo s tehnologijo *Node.js*, predvsem kot možnosti za hitro vzpostavitev spletnega strežnika za potrebe lokalnega testiranja aplikacij na napravi za sprejem IP-TV (*angl. set top box, STB*), ki jih je po specifikaciji razvijal zunanji izvajalec. Po kratki



Slika 1.1: Standardizacija in mejniki v razvoju JS in Node.js.

proučitvi zahtev smo za spletni strežnik izbrali *Express.js*, ker omogoča tudi enostaven razvoj programskih vmesnikov REST. Strežnik *Express.js* je tako služil, kot testni sistem za generiranje različnih možnih kombinacij dokumentov JSON, ki so služile sprotnemu preverjanju implementacije aplikacije za STB, preverjanju ponovljivosti zaznanih napak in ob vsakem danem popravku aplikacije tudi preverjanje le-tega.

Vzporedno je druga skupina razvijalcev razvijala produkcijski API vključno s sistemom upravljanja vsebin (*angl. content managment system, CMS*). Ta je bil razvit v ustaljenem razvojnem okolju kot aplikacija *MVC* implementirana v jeziku C# in ogrodju .Net. V grobem lahko ocenimo, da razvoj enakovrednih REST APIjev lahko zahteva nekaj več človek-ur od razvoja na osnovi Node.js in spletnega strežnika *Express.js*.

1.2.2 Orodje za povezljivost tehnologij

Naše spoznavanje s skladom tehnologij *MEAN* se je tudi nadaljevalo na projektu, kjer smo izbrali *Node.js*, kot programski jezik za reševanje problematike povezljivosti različnih tehnologij in zopet možnosti enostavne implementacije *REST APIjev*, na osnovi *Express.js*. Aplikacija implementirana *Node.js* je tako služila kot preprost agent, ki je spremljal skrbniški vmesnik strežnika *RabbitMQ* in v rednih časovnih intervalih sledil pojavom novih čakalnih vrst ter se s pomočjo vmesnikov modula *node-amqp* [15] naročal na poslušanje dogodkom protokola AMQP [16] na čakalnih vrstah in s tem zajem podatkov iz čakalnih vrst ter vstavljanje le-teh v podatkovno zbirko *Oracle MySQL*. Agent napisan v *Node.js* se je izkazal kot učinkovita rešitev.

1.2.3 Razvoj programskih vmesnikov

Na osnovi dobrih izkušenj in enostavnosti implementacije ob bogatem ekosistemu odprtokodnih modulov, je bila izbira strežnika za potrebe razvoja odjemalske aplikacije ponujena na pladnju. In za vmesnike REST API je bila izbrana tehnologija *Node.js* v kombinaciji s strežnikom *Express.js*.

Pri večjih projektih je za bolj učinkovit razvoj vmesnikov API smiselno poleg uporabe *Express.js* vpeljati še dodatna orodja in ogrodja, kot so specifikacije *Swagger*, kot ključni del specifikacije *The OpenAPI Specification* v okviru iniciative *OAI* [17] in odprtokodno ogrodje za razvoj aplikacijskih vmesnikov *LoopBack* [18].

1.2.4 Strežniško generiranje spletnih strani

Spoznavanje s spletnim strežnikom *Express.js* se je nadaljevalo tudi na vzporednem projektu, kjer smo tekom tedna implementirali enostavnega agenta temelječega na klicih *REST APIja* za beleženje dohodnih in odhodnih klicev manjšega klicnega centra. Aplikacija klicnega centra je ob posameznih dogodkih klicala *REST API* in prožila odprtje novega vnosa v sistemu upravljanja odnosov s strankami (*angl. customer relationship management, CRM*): ustrezen zapis v podatkovno zbirko in prikaz ustreznih vnosnih polj v spletnem brskalniku agenta v klicnem centru. Strežnik *Express.js* je v našem primeru spletne strani generiral s pomočjo predlog *Pug* 2.2.

Tako zasnovan enostaven *CRM* se je izkazal, kot zelo učinkovita rešitev, ki uspešno pokriva vse osnovne nezahtevne zahteve manjšega klicnega centra in tako precej nižje začetne stroške od nakupa in implementacije kompleksnejše komercialne rešitve. Tekom pol leta uporabe so se pojavile predvsem težave vezane na pripravo poročil. Po analizi smo lahko dodatne optimizacije enostavno implementirali na klicih poizvedb *SQL*. Dodatno smo se izognili podvajanju izdelave zgodovinskih poročil, ki smo jih predhodno odlagali v imenik s statičnimi datotekami, ki jih strežnik *Express.js* streže neposredno.

1.2.5 Hramba podatkov

Podobno so se tudi na prvem projektu pojavile težave na nivoju podatkovne zbirke, navkljub skrbni izbiri stabilnega in preverjenega modula za dostop do podatkovne zbirke MySql in uporabi ustreznega deljenja DB-povezav (*angl. connection pooling*). V želji po nadaljnji optimizaciji aplikacij smo začeli

raziskovati možnosti uporabe alternativnih tehnologij.

Iz vidika večje fleksibilnosti napram klasičnim *RDBMS* se nam kot naravna izbira ponuja podatkovna zbirka *MongoDB*, pri čemer je navdušujoča možnost hrambe dokumentov v notaciji *JSON* brez vnaprej določene sheme (*angl. schema-free*).

Tudi učinkovitost in zmogljivost sta v prid MongoDB proti MySQL: MongoDB lahko dosega do 50-krat hitrejša vstavljanja in obenem dosega skoraj 3-krat boljše čase pri enostavnih in kompleksnih bralnih poizvedbah [19]. MongoDB omogoča tudi enostavno širitev navzven (*angl. scaling out*), saj omogoča razcepljanje podatkov med več strežniki in samodejno prerazporejanje dokumentov [20].

Manj navdušujoče je dejstvo, da skrbniki podatkovnih zbirk (*angl. database administrator, DBA*), vajeni predvsem relacijskih podatkovnih zbirk, niso pripravljeni na nove, alternativne tehnologije. Kar je razumljivo, saj vsaka nova tehnologija zahteva dodatna znanja in izobraževanja, v katera podjetja vlagajo pogosto premalo ali praktično nič. Stanju pripomore dejstvo, da so uveljavljeni ponudniki programskih rešitev in tehnologij dobro vpeti v ustaljene posle. Z drugimi besedami, odgovornim za tehnologijo je lažje shajati z obstoječimi, dobro vpeljanimi tehnologijami, kot prevzemati izzive preoblikovanja in tehnološke evolucije.

Spoznavanje novih tehnologij je tako večinoma prepuščeno lastni iniciativi posameznikov, ki v svojem prostem času spremljajo dogajanje v industriji in pojav novih trendov.

1.2.6 Prototipni razvoj

Osnovni namen vsakega prototipa je zgodnja identifikacija in rešitev nejasnosti. Za potrebe prototipnega razvoja izberemo sklad tehnologij, ki omogoča hitro implementacijo delujočega prototipa aplikacije. Programski prototip je delna ali možna implementacija predlaganega novega izdelka [21] in služi trem glavnim namenom:

- razjasnitvi in dopolnitvi zahtev,

- raziskavi alternativnih rešitev in
- rasti v končni izdelek.

Prototipe ločimo na horizontalne prototipe in vertikalne prototipe. Prvi služijo prikazu funkcionalnih možnosti in jih poznamo kot makete (*angl. mock-up*). Vertikalne prototipe poznamo tudi kot preskus koncepta (*angl. Proof of Concept, PoC*) in delujejo kot resnični sistemi na vseh plasteh. Druga možna delitev prototipov je delitev na raziskovalne prototipe (*angl. Throwaway Prototype*) in evolucijske prototipe.

Tako sklad *MEAN* kot ogrodje *Meteor* lahko predstavljata odlični orodji za prototipni razvoj. Služita lahko tudi za razvoj evolucijskih prototipov, ki bodo tekom cikla prerasli v končni izdelek. Meteor je tako primerna izbira za vertikalni evolucijski prototip mobilne aplikacije.

1.3 Metodologija ocenjevanja ustreznosti

Ustreznost posamezne tehnologije, sklada tehnologij in ogrodja bomo ocenjevali glede na dano problemsko področje. Vsak problem bomo najprej razdelili na manjše ter nato za vsak podproblem ocenili primernost dane tehnologije, sklada tehnologij ali ogrodja. Pri tem bomo uporabili oceno od 1 do 5. Končna ocena je vsota posameznih ocen, ki jo bomo pomnožili še z 2, če je izbrana tehnologija odprtokodna.

Za ocenjevanje in primerjavo odprtokodnih tehnologij in modulov JS, ki so dostopni na shrambah kode *Git* spletnega mesta GitHub [22], bomo za oceno skupnosti uporabili formulo (1.1). Za izračun bomo uporabili javno dostopna števila zvezdic (*angl. stars*), števila razcepov kode (*angl. forks*), števila objav kode (*angl. commits*) in števila sodelujočih razvijalcev (*angl. contributors*). Za pomoč pri izračunu smo si v ogrodju Meteor pripravili aplikacijo.

$$Ocena_{(skupnost)} = stars + forks + commits + (contributors \times 10) \quad (1.1)$$

Pri navajanju izračunov bomo oceni v oklepaju (skupnost) pripisali ime avtorja in ime repozitorija ter tako bralcem omogočili, da podatke preverijo na spletnem mestu GitHub.

Poglavje 2

Sklad tehnologij MEAN

Sklad tehnologij MEAN (*MongoDB*, *Express.js*, *Angular*, *Node.js*) se pojavi predvsem, kot sodobna alternativa skladu tehnologij *LAMP* (*Linux*, *Apache*, *MySQL*, *PHP*), ki je bil kot tak poimenovan leta 1998.

Že po ključnih komponentah hitro opazimo, da operacijski sistem ni več del sodobnega sklada tehnologij. Temu je tako predvsem zaradi vedno širše uporabe možnosti izvajanja v oblaku, kjer uporabnik zakupi določene sistemske zmogljivosti, operacijski sistem je tako lahko vse bolj prikrit, neprisoten in virtualen.

Sklad *MEAN* je odprt in podprt na vseh vodilnih sistemih v oblaku in vseh sodobnih operacijskih sistemih, čeprav se ob sami uporabi lahko dejansko izkaže, da je uporaba in konfiguracija samega sklada običajno lažja na sistemih UNIX. Težave pridejo še toliko bolj do izraza na ne-UNIX operacijskih sistemih, ko paketi *Node.js* vsebujejo tudi odprtokodne komponente, ki niso napisane v JS in zahtevajo predhodno prevajanje oz. gradnjo komponent z različnimi prevajalniki in sistemi gradnje, ki so še vedno vezani na sam operacijski sistem. Nevarnosti migracije razvijalcev sodobnih tehnoloških skladov na operacijske sisteme UNIX, predvsem odprtokodne različice Linux in komercialni OS X, se končno prilagaja tudi družba Microsoft, ki je med drugim najavila podporo ukazni vrstici z lupino *Bash* v operacijskem sistemu Windows. Najava je predvsem odziv na povratne informacije razvijalcev, ki

jim sistem Windows otežuje uporabo odprtokodnih orodij [23].

MEAN ni na voljo kot celovito ogrodje z enotno namestitvijo, zato si v enačbah (2.1), (2.2), (2.3), (2.4) in (2.5) najprej poglejmo ocene skupnosti posameznih tehnologij:

$$Ocena_{(mongodb/mongo)} = 9375 + 2639 + 34162 + 252 \times 10 = 48696 \quad (2.1)$$

$$Ocena_{(expressjs/express)} = 25578 + 4792 + 5244 + 189 \times 10 = 37504 \quad (2.2)$$

$$Ocena_{(angular/angular.js)} = 49690 + 23966 + 7820 + 1476 \times 10 = 96236 \quad (2.3)$$

$$Ocena_{(angular/angular)} = 12512 + 3254 + 4514 + 278 \times 10 = 23060 \quad (2.4)$$

$$Ocena_{(nodejs/node)} = 23827 + 3450 + 14134 + 930 \times 10 = 50711 \quad (2.5)$$

Sicer je vsaj nekaj aplikacijskih ogrodji osnovanih na tehnologijah MEAN. Poglejmo si nekaj ključnih predstavnikov in v enačbah (2.6), (2.7) in (2.8) njihove ocene skupnosti:

- mean.io

$$Ocena_{(linnovate/mean)} = 8998 + 2732 + 1715 + 168 \times 10 = 15125 \quad (2.6)$$

- sailsjs.org

$$Ocena_{(balderdashy/sails)} = 14622 + 1529 + 5686 + 206 \times 10 = 23897 \quad (2.7)$$

- ionicframework.com

$$Ocena_{(driftyco/ionic)} = 24133 + 4807 + 4428 + 259 \times 10 = 35958 \quad (2.8)$$

Iz ocen skupnosti je razvidno, da imajo posamezne tehnologije MEAN boljše ocene od zgoraj naštetih ogrodij. Izjema je Angular 2 (2.4), ki ima tudi nižjo oceno od prve različice Angular.js (2.3), kar nakazuje razmeroma nizek sprejem (*angl. adoption*) med razvijalci. Kot izhodišče pri izbiri primernosti posameznega ogrodja lahko priporočimo ogrodja, ki dosegajo ali presegajo oceno skupnosti Express.js (2.2). Temu pogoju se najbolj približa ogrodje *ionicframework.com*.

2.1 MongoDB

Na podatkovno zbirko *MongoDB* lahko gledamo tudi, kot prvi nivo sklada tehnologij *MEAN*. MongoDB je predstavnik podatkovnih shramb *NoSQL*, družine podatkovnih baz, ki ne uporabljajo (samo) poizvedb *SQL*.

MongoDB je dokumentna podatkovna shramba, ki kot osnovo uporablja strukture *JSON* pretvorjene v binarni zapis *BSON* 2.1.1, kar je lahko prednost za celosten razvoj odjemalsko-strežniških aplikacij. Na same dokumente v podatkovni shrambi lahko gledamo kot analogijo strukturam programskih jezikov, ki omogočajo asociacijo ključa in vrednosti (*angl. key-value*). Predstavniki tovrstnih podatkovnih struktur so imeniki (*angl. dictionaries*), razpršilne tabele (*angl. hashes*), mapiranje (*angl. maps*) in asociativna polja (*angl. associative arrays*).

MongoDB hrani dokumente v zbirkah (*angl. collections*) (Slika 2.1), ki predstavljajo analogijo tabelam relacijske podatkovne baze, zbirka je tako skupina povezanih dokumentov, ki imajo nabor skupnih indeksov.

JavaSkriptna objektna notacija *JSON* se lahko tako propagira od podatkovne zbirke do samih odjemalcev. Dejansko se JSON tekom sklada tehnologij lahko pretvarja iz JSONa v binarni zapis BSON in obratno.

Pomembno je, da so pretvorbe (*angl. serialization*) dobro specificirane in standardizirane. Kot primer, vzemimo pretvorbo datuma v niz znakov z metodo `JSON.stringify( value )` oz. `Date.prototype.toJSON( key )` , kjer ES5.1 [11] določa pretvorbo v datumski zapis ISO 8601 [24]. Vidimo lahko, da se z upoštevanjem standardov izognemo situacijam, kot se lahko pojavljajo v drugih tehnologijah [25, 26].

2.1.1 BSON

BSON ali *Binarni JSON* je serijski binarni format, ki se uporablja za hrambo dokumentov. Zasnova binarnega formata BSON omogoča naslednje značilnosti: lahkotnost (*angl. lightweight*) z minimalnimi režijskimi podatki, prehodnost (*angl. traversability*) in učinkovitost [27]. Knjižnice za binarni format BSON

na voljo za večino programskih jezikov. V Node.js lahko uporabimo uradno knjižnico *node-mnongodb-native* ali *node-buffalo*. V enačbah (2.9) in (2.10) si poglejmo njuni oceni skupnosti.

$$Ocena_{(mongodb/node-mongodb-native)} = 4766 + 1026 + 3719 + 214 \times 10 = 11651 \tag{2.9}$$

$$Ocena_{(marcello3d/node-buffalo)} = 101 + 21 + 47 + 4 \times 10 = 209 \tag{2.10}$$

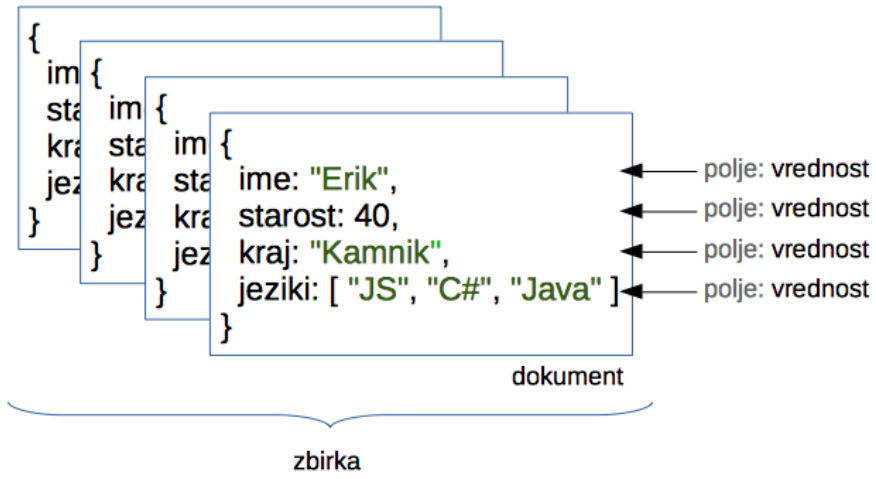
Na knjižnici *node-buffalo* ne poteka aktiven razvoja, kar se odraža tudi v naši oceni skupnosti (2.10). Tako je večinoma smiselna uporaba knjižnice *node-mnongodb-native* za katero skrbijo avtorji MongoDB.

2.1.2 Poizvedbe v MongoDB

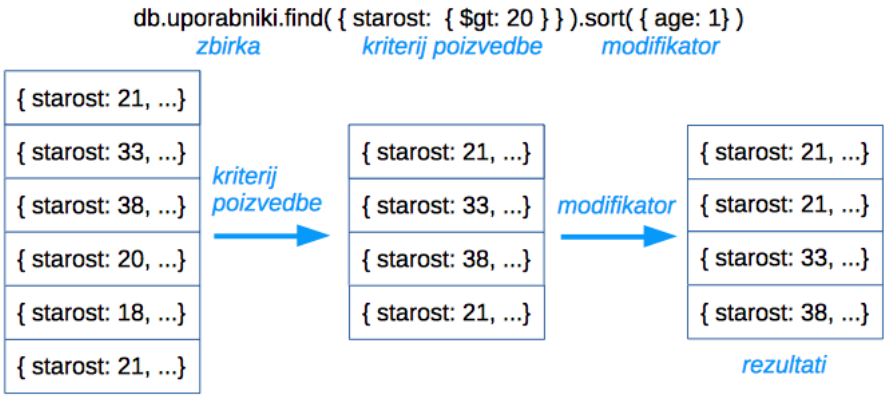
Poizvedbe izvajamo s pomočjo metod MongoDB za operacije CRUD, ki si jih lahko ogledamo v Tabeli 2.1. Primer bralne poizvedbe (Slika 2.2) uporablja metodo `db.imezbirke.find( {} );` , kjer kriterije poizvedbe podajamo kot JSON, rezultat nato še uredimo z metodo `.sort()` .

MongoDB, poleg enostavnih operacij kot so štetja `.count()` , enoličnosti `.distinct()` , grupiranja `.group()` , omogoča tudi operacije *Map-Reduce* `db.collection.mapreduce(mapirnaFunkcija, redukcijskaFunkcija, opcije);` in uporabo agregacijskega cevovoda (*angl. aggregation pipeline*) `db.collection.aggregate([ faza1, faza2, ... fazaN ]);` .

Primerjavo agregacij MongoDB [28] s stavki SQL (*angl. SQL clauses*) najdemo v Tabeli 2.2.



Slika 2.1: Prikaz hrambe podatkov v dokumentih in zbirkah MongoDB.



Slika 2.2: Prikaz delovanja enostavne bralne poizvedbe v MongoDB.

Tabela 2.1: Metode MongoDB za operacije vstavljanja, branja, posodabljanja in brisanja.

Operacija	Metoda MongoDB
Vstavljanje	db.collection.insert() db.collection.insertOne() db.collection.insertMany()
Branje	db.collection.find()
Posodabljanje	db.collection.update() db.collection.updateOne() db.collection.updateMany() db.collection.replaceOne()
Brisanje	db.collection.remove() db.collection.deleteOne() db.collection.deleteMany()

Tabela 2.2: Primerjava stavkov SQL in agreagacij MongoDB.

Stavki SQL	Agregacije MongoDB
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum
JOIN	$lookup

2.2 Express.js

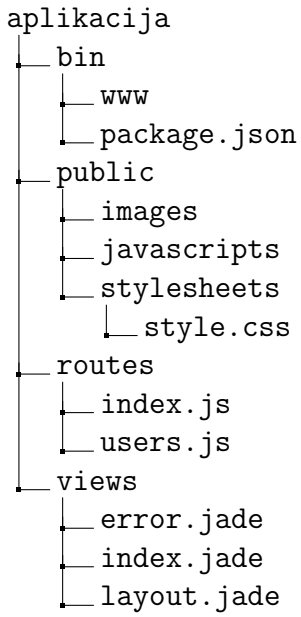
Express.js je minimalistično spletno ogrodje (*angl. web framework*), ki se izvaja kot aplikacija Node.js. Omogoča robusten nabor ključnih funkcionalnosti za spletne in mobilne aplikacije [29], vsebuje vse potrebne metode HTTP in vmesno opremo (*angl. middleware*), ki omogočajo enostavno in hitro gradnjo robustnih APIjev. Express je projekt fundacije *Node.js Foundation*. Za namestitev spletnega ogrodja Express.js potrebujemo predhodno nameščen Node.js, saj namestitev izvedemo s pomočjo ukaza npm:
`npm install express -g` .

Ko imamo na svojem sistemu nameščen Express.js, lahko novo aplikacijo preprosto generiramo s pomočjo ukaza, ki ga kličemo v delovnem imeniku: `express aplikacija` . Generator bo v naši delovni mapi ustvaril novo mapo *aplikacija* in v njej pripravil strukturo aplikacije Express. Struktura, ki jo pripravi generator aplikacij Express.js (Slika 2.3), je le ena od možnih struktur projekta spletne aplikacije, ki si jo lahko po potrebi prilagodimo lastnim željam in potrebam.

Zatem nam preostane le še, da v mapi aplikacija poženemo `npm install` in s tem namestimo vse zahtevane module. Kateri moduli so vsebovani lahko preverimo v datoteki *package.json*, ter nato poženemo aplikacijo z ukazom `npm start` , ter obiščemo spletno stran v brskalniku na naslovu:
`http://localhost:3000/`.

S tem imamo na voljo povsem delujoči spletni strežnik. Statične vsebine lahko odlagamo v mapo *public*, običajno so to slike, minimirane odjemalske skripte JS in datoteke kaskadnih slogov CSS. Spletne strani Express.js privzeto generira s pomočjo sistema predlog *Pug* (včasih znanem po imenu *JADE*). Na voljo imamo tudi druge sisteme predlog, kot sta *Mustache* ali *EJS*.

Izbira sistema predlog je prepuščena razvijalcu, priljubljenost posameznih predlog ocenimo na podlagi ocene skupnosti na GitHubu v enačbah (2.11), (2.12) in (2.13).



Slika 2.3: Struktura nove aplikacije Express.

$$Ocena_{(pugjs/pug)} = 11192 + 1566 + 2419 + 175 \times 10 = 16927 \quad (2.11)$$

$$Ocena_{(janl/mustache.jsg)} = 9363 + 1852 + 694 + 86 \times 10 = 12769 \quad (2.12)$$

$$Ocena_{(mde/ejs)} = 660 + 101 + 2828 + 88 \times 10 = 4469 \quad (2.13)$$

Iz ocen skupnosti je razvidno, da je privzeti sistem predlog *Pug* tudi najbolj priljubljen. Sledi mu sistem predlog *Mustache*. Najmanj priljubljen je sistem predlog *EJS*.

2.3 Angular.js

Angular.js je odprtokodno ogrodje za spletnih in mobilnih aplikacij, ki cilja na poenostavitev razvoja in testiranja aplikacij s pomočjo arhitektur *MVC* in *MVVM* oz. *MV-karkoli* (*angl. model-view-whatever, MVW*). Temelji na prepričanju, da je deklarativno programiranje primerno za UI in povezavo programskih komponent, medtem ko je imperativno programiranje primernejše za poslovno logiko aplikacije [30].

Ogrodje Angular razširja HTML in omogoča *dvosmerno povezovanje podatkov* (*angl. two-way data-binding*) 2.3.2, kar omogoča samodejno sinhronizacijo modelov (*angl. model*) in pogledov UI (*angl. views*). Angular je ogrodje za čelni del sistema (*angl. front-end framework*), ki omogoča izgradnjo dinamične spletne strani *SPA*. Angular pomaga organizirati kodo JS in pomaga graditi odzivne (*angl. responsive*) in hitre spletne strani ter obenem omogoča enostavno testiranje. Angular kot odjemalsko ogrodje JS omogoča dodajanje interaktivnosti spletnim stranem HTML. Podrobneje si poglejmo nekaj gradnikov in izrazoslovje, ki ga uporablja ogrodje.

2.3.1 Gradniki in izrazoslovje

Direktive

Angular spletnim stranem HTML dodaja *obnašanje* (*angl. behaviour*) s pomočjo *direktiv* (*angl. directives*). Direktive so Anglularju lastni atributi na elementih HTML s predpono `ng-` , ki Angularju povedo, katera koda JS naj se izvede ali sklicuje. Atribut `ng-controller="StoreController"` na elementu `body` , tako izvede klic funkcije `function StoreController()` v datoteki *app.js*.

Moduli

Moduli (*angl. modules*) vsebujejo dele aplikacije Angular.js in omogočajo lažje vzdrževanje, testiranje in boljšo preglednost aplikacije. Modul ustvarimo s pomočjo metode `angular.module` , kot je prikazano v primeru Kode 2.1. V samih modulih lahko uporabljamo tudi druge module. Z moduli definiramo in oblikujemo tudi odvisnosti aplikacije (*angl. application dependencies*). Modul je kontejner za krmilnike. Krmilniki vedno pripadajo modulu.

```
1  var app = angular.module('imeaplikacije', [ ]);
```

Koda 2.1: Primer ustvarjanja novega modula v datoteki *app.js*.

Krmilniki

Krmilniki (*angl. controllers*) definirajo obnašanje aplikacije s pomočjo definicije funkcij in vrednosti. Direktiva `ng-controller` definira krmilnik, ki je objekt JS ustvarjen s konstruktorjem objekta JS.

Doseg

Doseg (*angl. scope*) je vezni del med vmesnikom UI in krmilnikom JS. Doseg podajamo z argumentom `$scope` .

Servisi

Servisi (*angl. services*) so funkcije ali objekti, ki so na voljo v aplikaciji Angular.js. Angular vsebuje kar nekaj internih servisov: `$http` , `$route` , `$window` , `$location` , itd. Lastne servise lahko definiramo s pomočjo metod `factory()` ali `service()` [31].

Izrazi

Izrazi (*angl. expressions*) omogočajo vstavljanje dinamičnih vrednosti v HTML, zapišemo jih v dvojnih zavitih oklepajih: `{{4+6}}` . Izrazi pridejo do izraza v kombinaciji z direktivami, npr. direktivo za ponavljanje:

```
<h1 ng-repeat="x in records">{{x}}</h1>
```

Filtri

Filtri (*angl. filters*) oblikujejo vrednosti izrazov za prikaz v UI. Filtre apliciramo na izraz s sintakso: `{{ izraz | imefiltra:opcije }}` . Angular.js vsebuje naslednje filtre:

- `currency` oblikuje število v valutni zapis.
- `date` omogoča oblikovanje datumov.
- `filter` se uporablja za izbor podmnožice polja.
- `json` omogoča pretvorbo objekta v niz JSON.
- `limitTo` omeji polje ali niz na določeno število elementov ali znakov. Npr. `ng-repeat="products in store.products | limitTo:3"` .
- `lowercase` pretvori niz znakov v male črke.
- `number` oblikuje število v niz.
- `orderby` uredi polje.
  Npr. `ng-repeat="products in store.products | orderBy:'-price'"` .
- `uppercase` pretvori niz znakov v velike začetnice.

2.3.2 Dvosmerno povezovanje podatkov

Dvosmerno povezovanje podatkov (*angl. two-way data binding*) predstavlja sinhronizacijo med modelov in pogledom UI. Predloge v Angularju delujejo drugače. Najprej se predloga prevede v brskalniku. Rezultat prevajanja je živ pogled UI (*angl. live view*), kakršna koli sprememba v njem se takoj odseva v model in kakršne koli spremembe na modelu se propagirajo na živ pogled UI. Model predstavlja enoten vir resnice (*angl. single-source-of-truth*) za stanje aplikacije [32], kar poenostavlja programski model. Na prikaz UI lahko gledamo kot na projekcijo modela v realnem času.

2.3.3 Vstavljanje odvisnosti

Angular ima vgrajen podsistem *vstavljanja odvisnosti* (*angl. dependency injection, DI*), kar olajša razvoj, razumevanje in testiranje aplikacije. DI omogoča povpraševanje po odvisnostih in s tem je za dostop do ključnih servisov Angular dovolj, da servis dodamo kot parameter, Angular bo zaznal, da ga potrebujemo ter vrnil primerek servisa.

2.4 Node.js

Node.js ® je strežniško JS okolje (*angl. environment*) osnovano na zmogljivem odprtokodnem JS-interpreterju V8, razvitem v programskem jeziku *C++* pod okriljem projekta *Chronium*. V8 implementira ES3 specifikacije ECMA-262 in s pomočjo vgrajenega prevajalnika (*angl. Just-In-Time, JIT*) zagotavlja hitro izvajanje kode JS.

Poglejmo si izračun ocene skupnosti V8 (2.14).

$$Ocena_{(v8/v8)} = 2458 + 624 + 10000 + 118 \times 10 = 14262 \tag{2.14}$$

V8 je večinoma vključen v večje projekte, kot so brskalniki, Node.js ali ogrodje *Electron*. Povprečni razvijalci tako z V8 nimajo neposrednega stika in to se odseva v nizki oceni skupnosti (2.14).

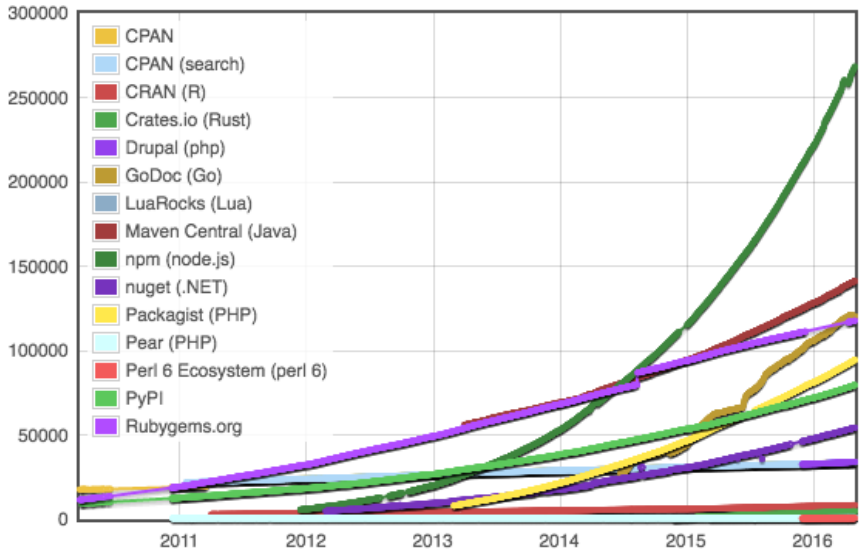
Node je v celoti zasnovan za asinhrono izvajanje V/I operacij. Tudi druga okolja omogočajo asinhrone V/I funkcionalnosti a so le-te večinoma omejene v asinhronih različicah. Po drugi strani je v Node.js vse, če smo bolj natančni skoraj vse, asinhrono. Pisanje ne-asinhrone kode v Node.js je lahko težje od uporabe asinhrone kode. Kljub številnim razpravam, ali je asinhrono programiranje pravi vademekum, ostaja dejstvo, da je asinhronost lahko zelo primerna rešitev za številne spletne in omrežje probleme, ki tudi sami po sebi vsebujejo asinhronost.

S programiranjem v JS se lahko tudi izognemo nepotrebnim miselnim preklopom in s tem tudi nepotrebnim sintaktičnim napakam pri programiranju v različnih programskih jezikih. Razvijalec programske opreme se lahko tako z izbiro enega jezika, ki je primeren tako za razvoj na odjemalcu kot strežniku, bolje posveti in poglobi svoje znanje predvsem na enem ključnem programskem jeziku.

2.4.1 Npm

Sistem upravljanja paketov *Npm* je pomemben prednostni faktor Node.js. Rast Npm in primerjavo z ostalimi shrambami kode lahko vidimo na grafu na Sliki 2.4. Prav bogat in hitro rastoč nabor odprtokodnih paketov je ključna prednost za izbiro tehnologij, ki temeljijo na JS in Node.js. Razvijalec lahko svojo lastno rešitev sestavi iz številnih že obstoječih odprtokodnih gradnikov, kar si lahko predstavljamo kot sestavljanje s pomočjo kock. Razvijalci lahko izvirne rešitve in izboljšave preko Npm prispevajo nazaj skupnosti, ta tako že presega četrt milijona paketov. Npm je največji ekosistem odprtokodnih knjižnic na svetu [33, 34].

Npm je po številu paketov že sredi leta 2014 presegel *Apache Maven* [34], shramba kode programskega jezika Java. Pri tem moramo upoštevati, da se Maven uporablja tudi za razvoj aplikacij za naprave Android, ki po ocenah IDC presega 80% tržni delež pametnih telefonov [35].



Slika 2.4: Rast števila prenosov modulov za različne programske jezike [34].

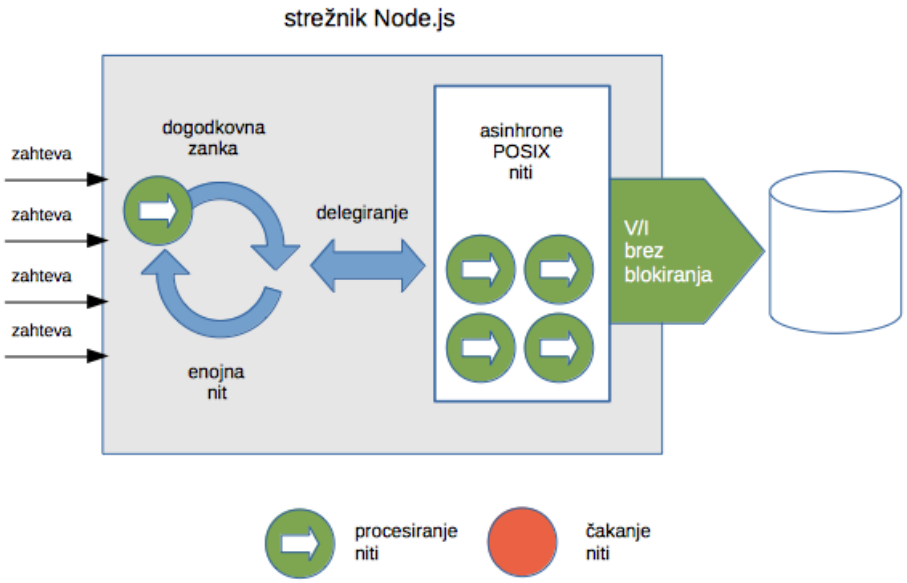
V enačbi (2.15) si poglejmo še izračunano oceno skupnosti Npm.

$$Ocena_{(npm/npm)} = 9498 + 1966 + 6852 + 368 \times 10 = 21996 \tag{2.15}$$

Glede na velikost samega ekosistema Npm je ocena skupnosti presenetljivo nizka, kar lahko pripišemo dejstvu, da je Npm sestavni del Node.js.

2.4.2 Dogodkovna zanka

Dogodkovna zanka (*angl. event loop*) okolju Node.js omogoča obvladovanje visokega števila hkratnih zahtev čeprav se izvaja v eni sami niti. Dogodkovna zanka delegira procesiranje operacijskemu sistemu preko vmesnika POSIX in medtem nadaljuje rokovanje novih zahtev in povratnih klicev, kot je prikazano na Sliki 2.5.



Slika 2.5: Dogodkovna zanka okolja Node.js.

Poglavje 3

Ogrodje Meteor

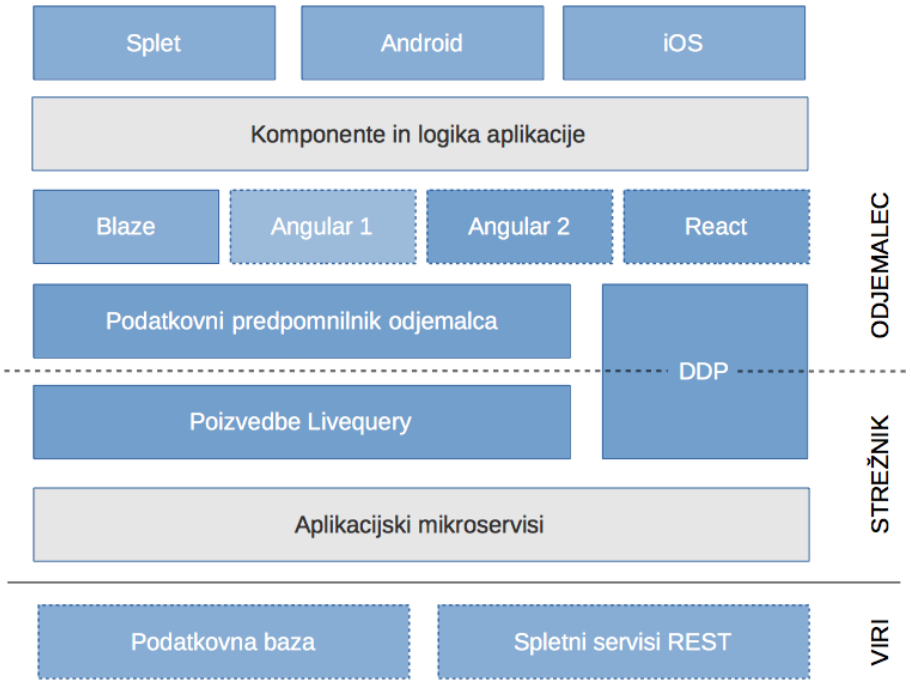
Meteor sestavlja nabor odprtokodnih projektov (*MIT licenca*), ki skupaj tvorijo celovito aplikacijsko platformo JS za reaktiven (*angl. reactive*) razvoj aplikacij na mobilnih platformah in spletu.

Ogrodje Meteor si lahko predstavljamo kot sklad (Slika 3.1), ki poteka od virov podatkov preko strežniškega dela aplikacije do odjemalcev. Na ogrodje je sestavljeno iz treh sklopov. Sklopa, ki skrbi za uporabniški vmesnik (UI), podatkovnega sklopa (*angl. data*) in orodij za gradnjo (*angl. build tools*). Sklop, ki skrbi za UI sestavljajo komponente:

- Angular, Blaze ali React,
- usmerjanje (*angl. routing*) in
- odjemalska reaktivnost (*angl. client reactivity*).

Podatkovni sklop sestavljajo:

- Odjemalski predpomnilnik *Minimongo*,
- tehnologija *WebSocket*,
- metode (*angl. methods*) in objave (*angl. publications*) ter
- gonilnik podatkovne baze.



Slika 3.1: Ogrodje Meteor kot sklad tehnologij

Orodja za gradnjo vsebujejo:

- podporo ES,
- mobilne kontejnerje,
- vroče vrivanje kode (*angl. hot code push*) in
- storitve nameščanja aplikacije na spletni strežnik (*angl. deployment*).

3.1 Primerjava MEAN in Meteor

Ob primerjavi sklada tehnologij MEAN z orodji in tehnologijami, ki jih vsebuje Meteor, ugotovimo precej podobnosti:

- oba temeljita na Node.js,

- v obeh igra ključno vlogo JS,
- oba nudita podporo za ES6,
- oba uporabljata MongoDB,
- oba nudita možnost uporabe Angular.js in Angular 2.

Šele razlike nam pravzaprav pokažejo dodatne prednosti, ki jih prinaša Meteor. Te lahko na kratko povzamemo v naslednjih točkah:

- MEAN nima enotne namestitve. Razvijalec mora tako namestiti posamične tehnologije in orodja. *Node.js*, *Bower*, *Grunt*, *Yeoman* so le nekatera orodja s pomočjo katerih gradimo aplikacije sklada tehnologij MEAN.
- Meteor vsebuje vse kar potrebujemo v eni enostavni namestitvi.
- Meteor že vsebuje delujoč MongoDB, ki nam je na voljo takoj, ko poženemo aplikacijo. Tudi dostop preko ukazne vrstice je poenostavljen s pomočjo ukaza: `meteor mongo` .
- Meteor zasede manj prostora na disku.
- Meteor že vključuje podporo za gradnjo mobilnih aplikacij s pomočjo ogrodja Apache Cordova 3.9.
- Meteor se poslužuje podatkov na žici (*angl. data on the wire*), kar pomeni, da strežnik pošilja podatke namesto dokumentov HTML, ki jih nato odjemalec ustrezno prikaže.
- Meteor omogoča *reaktivnost* na celotnem skladu, kar omogoča, da vmesnik vedno odseva pravo stanje z najmanjšim razvojnim vložkom.
- Meteor omogoča kompenzacijo za čas zakasnitve (*angl. latency compensation*). Tako so rezultati operacij vidni takoj, medtem ko pri običajnih aplikacijah uporabniki čakajo, da se določene operacije izvedejo na strežniku, preden se odsevajo na vmesniku.

- Meteor nudi višjo stopnjo abstrakcije in razvijalcem omogoča razvoj aplikacij brez skrbi o kompleksnosti povezljivosti med odjemalcem in strežnikom.

- Meteor ponuja lastno infrastrukturo v oblaku *Galaxy*, možna je tudi namestitev na storitve v oblaku drugih priljubljenih ponudnikov.

- Meteor minimira in veriži vse datoteke JS in CSS, ter jih streže kot statične datoteke.

- Snovalci ogrodja Meteor so mnenja, da je linearen model izvajanja bolj primeren za tipične strežniške programe. Meteor in MEAN se zato razlikujeta v načinu izvajanja strežniške kode (Koda 3.1).

- Meteor omogoča dobre prakse za mobilne spletne aplikacije W3C [36]. Združevanje in minimiranje JS in CSS, podvojevanje lokalnih podatkov na strežnik in predvsem zmanjšanje zaznanih zakasnitev (*angl. percieved latency*), kar je ključno za uporabnost (*angl. usability*) [37] in proces izboljšav interakcije uporabnik-izdelek (*angl. User Experience Design, UXD*) [38].

Poglejmo si še enačbo (3.1) za oceno skupnosti ogrodja Meteor.

$$Ocena_{(meteor/meteor)} = 34214 + 4220 + 10000 + 293 \times 10 = 51364 \qquad (3.1)$$

Na podlagi ocene skupnosti lahko sklepamo, da je ogrodje Meteor v kontekstu ogrodij za gradnjo mobilnih aplikacij bolj priljubljeno od ogrodja Ionic (2.8) in Apache Cordova (3.7). Podobno ima tudi boljšo oceno skupnosti v primerjavi s spletnim ogrodjem Express.js (2.2). Na račun prejetega števila zvezdic in razcepov kode ima tudi nekoliko boljšo oceno skupnosti pred Node.js (2.5). V kontekstu ogrodij UI je Meteor na tretjem mestu, pred njim sta React (3.5) in Angular.js (2.3), ki jih sicer lahko uporabimo tudi v samem ogrodju Meteor.

3.1.1 Izvajanje strežniške kode

Marsikateremu razvijalcu se programiranje v Node.js lahko zdi nepraktično zaradi težav s povratnimi klici (*angl. callback soup, angl. callback hell*) [39].

Sicer je na voljo kar nekaj tehnik, s katerimi se lahko v okolju Node.js ognemo povratnim klicem: vlakna (*angl. fibers*), obljube (*angl. promisses*) in generatorske sorutine (*angl. generator-based corutines*) so le nekatere.

Meteor pri izvajanju strežniške kode privzeto uporablja linearen model izvajanja na osnovi vlaken *node-fibers* in programske vmesnike implementira nivo višje. Vlakna *node-fibers* nudijo nivo abstrakcije za dogodkovno zanko (*angl. event loop*) in omogočajo sekvenčno izvajanje funkcij in s tem pisanje asinhrone kode brez povratnih klicev. Obstoja vlaken se nam ni treba zavedati. S tem Meteor omogoča najboljše iz obeh svetov: asinhrono učinkovitost Node.js in programiranje v sinhronem slogu.

Možnost pisanja Node.js aplikacij brez povratnih klicev je atraktivna, tudi za vse razvijalce, ki so prav zaradi povratnih klicev sovražili Node.js. Vendar je lahko po drugi strani prav to težava, saj večina modulov Node.js pridobljenih preko Npm, vsebuje prav asinhrone funkcije. Tudi za to je poskrbljeno s pomočjo metode `Meteor.wrapAsync` , ki asinhrono funkcijo ovije in pretvori v sinhrono in nam omogoča pisanje kode, ki je na videz sekvenčna, čeprav se v ozadju v dogodkovni zanki Node.js izvaja asinhrono. Sinhrono ovijanje si poglejmo še na primeru (Koda 3.1).

```
1  Meteor.methods({
2    ustvariUporabnika: function( uporabnik ) {
3      // sinhrono ovijanje asinhrone funkcije
4      var ustvariUporabnikaSync = Meteor.wrapAsync(
           ustvariUporabnikaAsync, novUporabnik );
5      // klic sinhrone funkcije
6      try{
7        var rezultat = ustvariUporabnikaSync({
8          ime: uporabnik.ime,
9          priimek: uporabnik.priimek
10       });
11       // obravnavamo rezultat sinhronega klica
```

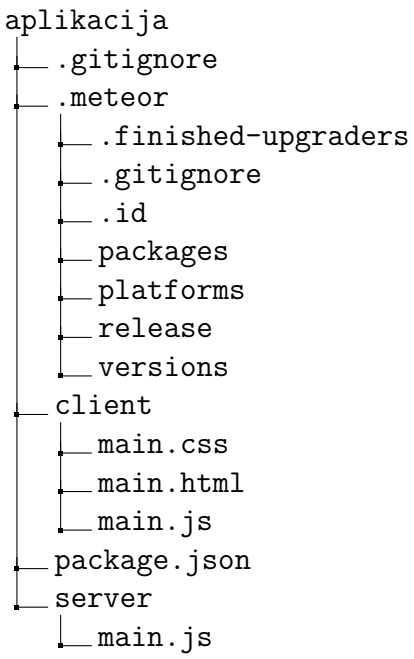
```
12        console.log("Rezultat :",rezultat);
13      }
14      catch(error){
15        console.log("Napaka: ",error);
16      }
17    }
18  });
```

Koda 3.1: Primer ovijanja asinhrone funkcije z metodo *Meteor.wrapAsync*.

3.2 Struktura map aplikacije Meteor

Meteor zbere vse datoteke JS v drevesni strukturi projekta z izjemo vsebine map *server*, *public* in *private*, jih minimira in streže vsakemu odjemalcu. Zato je zgolj naša lastna odločitev ali bomo vso kodo pisali v eno samo datoteko ali kodo ločili v posamezne datoteke. Poleg strukture (Slika 3.2), ki se ustvari ob klicu `meteor create aplikacija` , si poglejmo različne mape, ki jih lahko uporabimo poleg map *client* in *server*:

- **client** ... Datoteke JS v mapi *client* se izvajajo le na odjemalcih.
- **server** ... Mapa *server* vsebuje strežniško kodo JS.
- **private** ... Vsebina mape *private*, njenih podmap in datotek bo zasebna. Vsebina datotek je strežniku dostopna preko metod Assets API.
- **public** ... Meteor streže vse datoteke iz mape *public* kot statične datoteke. Mapa je tako namenjena slikam, ikonam in podobnim vsebinam.
- **imports** ... Datoteke v mapi in podmapah *imports* se strežejo z zakasnjenim nalaganjem (*angl. lazy-load*).
- **tests** ... Vse, kar je vsebovano pod mapo *tests* se ne nalaga. Vsebina te mape je namenjena kodi za testiranje.



Slika 3.2: Struktura nove aplikacije Meteor.

Struktura je pomembna zaradi pravil ob gradnji aplikacije. Datoteke v podmapah se nalagajo najprej, tako se datoteke v najgloblji mapi naložijo najprej in datoteke v korenski mapi naložene kot zadnje. Datoteke v posamezni mapi se nalagajo po abecednem vrstnem redu imen datotek. Po urejanju se vse datoteke vsebovane pod mapami imenovanimi *lib*, prestavijo pred vse ostale, pri čemer se vrstni red ohrani. V zadnjem koraku so vse datoteke, ki ustrezajo vzorcu *main.** premaknjene pred vse ostale, pri čemer se ohranja vrstni red.

3.3 Protokol DDP

DDP je protokol, ki ga Meteor uporablja za komunikacijo med odjemalcem in strežnikom. Vse izmenjave podatkov med objavami (*angl. publication*) na strežniku in naročnino (*angl. subscription*) na odjemalcu, klici metod in operacije MongoDB so posredovani kot sporočila DDP.

Čeprav se Meteor izvaja kot strežnik na enih vratih, interno deluje kot dva ločena strežnika: strežnik HTTP in strežnik DDP. Prvi služi streženju statičnih datotek in zahtev HTTP. Za ta namen se interno uporablja Node.js modul *connect*. Strežnik DDP skrbi za vse objave, operacije MongoDB in klice metod. Za to se uporablja transportni protokol *SockJS*. Lahko rečemo, da je strežnik DDP pravzaprav prilagojen strežnik SockJS.

Meteor ob vsaki zahtevi DDP ustvari novo vlakno *node-fiber*. Privzeto Meteor izvaja po eno zahtevo naenkrat za vsakega odjemalca, kar pomeni eno vlakno za vsakega odjemalca v danem trenutku, kar lahko tudi spremenimo. Čeprav MongoDB ni podatkovna zbirka v realnem-času, Meteor omogoča delovanje v realnem času s pomočjo:

- krožnega pozivanja (*angl. polling*) v 10-sekundnih intervalih in
- uporabo dnevnika operacij (*angl. oplog*).

Druga možnost je pomembna, ker je krožno pozivanje draga operacija.

3.4 Protokol HTTP

Meteor lahko opravlja tudi zahteve HTTP. Če želimo implementirati programske vmesnike REST (*angl. RESTful API*) [40, 41], moramo za uporabo funkcionalnosti protokola HTTP najprej dodati paket HTTP
z ukazom: `meteor add http` .

Paket je seveda tudi že vsebovan v paketih, kot sta npr. *REST for Meteor* ali *Restivus*, namenjenih implementaciji APIjev na ogrodju Meteor. V

enačbah 3.2 in 3.3 si poglejmo oceno skupnosti le-teh.

$$Ocena_{(stubailo/meteor-rest)} = 233 + 35 + 170 + 10 \times 10 = 538 \tag{3.2}$$

$$Ocena_{(kahmali/meteor-restivus)} = 410 + 61 + 154 + 11 \times 10 = 735 \tag{3.3}$$

Nizki oceni skupnosti lahko nakazujeta, da se večina razvijalcev odloči za paket HTTP. Ocene skupnosti paketa HTTP ne moremo pridobiti, ker ni na voljo kot samostojna shramba kode na spletnem mestu GitHub.

3.5 Procesor predlog Blaze

Blaze je privzet procesor predlog HTML za uporabniški vmesnik. Predloge Blaze uporabljajo preslednice (*angl. spacebars*) in so videti kot običajen HTML, ki vsebuje posebne oznake, vsebovane v dvojnih zavitih oklepajih `{{ }}` znotraj katerih navedemo bodisi spremenljivke JS, ki so dostopne v skripti, ali pomožne funkcije (*angl. helpers*), ki jih definiramo v sami kodi. Za kontrolo izvajanja v predlogi imamo na voljo konstrukte, kot so:

- Iteratorji `{{#each ... in ... }} ... {{else}} ... {{/each}}` , ki ponavljajo izpis bloka HTML za vsak element seznama in v primeru praznega seznama izpišejo vsebino, ki sledi `{{else}}` .
- Pogojno izvajanje `{{#if}} ... {{else}} ... {{/if}}` .
- S ključno besedo `{{#let ime=zaposleni.ime priimek=zaposleni.priimek }}` vpeljemo nove spremenljivke, in si pomagamo dostopati do posameznih atributov objekta ali izhodov pomožnih funkcij.
- In nenazadnje vključevanje drugih predlog `{{> mojaPredloga}}` .

Kot vidimo, je sama sintaksa predlog Blaze zelo intuitivna, kar omogoča hitro uporabo osnovnih konceptov, razvijalca pa tudi naravno usmerja v uporabo modularnosti. Blaze namenoma prikriva precejšnjo kompleksnost vnovičnih izrisovanj (*angl. re-rendering*), zato ni odveč pozornost razvijalca,

da poskrbi za nizko časovno zahtevnost svojih pomočniških funkcij in s tem tudi vnovičnih izrisovanj.

Za večje aplikacije je, zaradi boljše podpore modularnosti, priporočljiva izbira med tehnologijama Angular ali React. Poglejmo si še enačbo (3.4) za oceno skupnosti.

$$Ocena_{(meteor/blaze)} = 82 + 7 + 7391 + 115 \times 10 = 8630 \tag{3.4}$$

Če oceno skupnosti primerjamo z ocenami ostalih sistemov predlog, ugotovimo, da se uvršča na predzadnje mesto. Nizko oceno lahko pripišemo navezavi na ogrodje Meteor. Blaze je odlična izbira za hitro gradnjo aplikacije ali pripravo prototipov.

3.6 Podpora Angular.js

Če želimo, lahko namesto Blaze uporabimo Angular.js. Najprej odstranimo Blaze z ukazom: `meteor remove blaze-html-templates` , nato dodamo Angular.js z ukazom: `Meteor add angular-templates` in namestimo pakete npm z ukazom: `meteor npm install --save angular angular-meteor` .

Preden poženemo Meteor, poženemo urejevalnik Atom z ukazom `atom .` in v pripravljenem osnutku projekta zamenjamo nekaj datotek HTML, JS in CSS, ki so zasnovane za Blaze, s kodo za Angular.js prikazano v primerih Kode 3.2 in 3.3.

```
1  <head>
2    <title>Meteor aplikacija z Angular.js</title>
3  </head>
4  <body>
5    <div class="container" ng-app="moja-aplikacija">
6    Pozdrav iz Meteorja z Angular JS.
7    </div>
8  </body>
```

Koda 3.2: Prilagoditev kode *client/main.html* za uporabo Angular.js.

```
import angular from 'angular';
import angularMeteor from 'angular-meteor';

angular.module('moja-aplikacija', [
  angularMeteor
]);
```

Koda 3.3: Prilagoditev kode *main.js* za uporabo Angular.js.

Pred nami imamo osnutek Angular aplikacije v Meteorju. Dodajati lahko začnemo komponente Angular.js, ki jih bomo dodali v mapo, ki jo bomo pripravili za vsako posamezno komponento v mapi *imports/components*.

Struktura map Angular.js

Za organizacijo strukture map imamo na voljo dva pristopa:

- Urejanje po tipu datotek, kjer uporabimo mapo *controllers* za krmilnike in mapo *views* za uporabniški vmesnik.
- Urejanje glede na funkcionalnost, kjer določimo imena map. Npr. uporabniki, izdelki, itd.

Prvi pristop je primeren predvsem za manjše aplikacije, drugi se bolje obnese za večje aplikacije. Ker v Meteorju operiramo z zbirkami MongoDB, je zaradi sovpadanja boljša izbira strukture glede na funkcionalnost.

3.7 Podpora Angular 2

Meteor omogoča tudi možnost uporabe tehnologije Angular 2. Ta poleg možnosti razvoja v ES5 in ES6 podpira tudi jezika *TypeScript* in *Dart*, slednji je nov odprtokodni objektno orientiran programski jezik. Pred uporabo se oba jezika najprej prevedeta v JS, ki ga razumejo vsi sodobni brskalniki. Razvijalci ogrodja Angular 2 priporočajo uporabo programskega jezika TypeScript, zato večino dokumentacije, primerov in izobraževalnih vsebin najdemo predvsem v TypeScriptu.

Razlogi za prehod na Angular

Razlogov in prednosti za prehod iz tehnologije predlog Blaze na Angular je kar nekaj:

- Angular omogoča odjemalske aplikacije z arhitekturo MVW (MVC, MVVM).
- Ogrodje Angular je bolj zrelo iz vidika stabilnosti in funkcionalnosti.
- Angular obsega širšo skupnost razvijalcev in s tem je na voljo boljša podpora in več knjižnic.
- Omogoča uporabo ogrodja *Ionic*, drugega najbolj priljubljenega ogrodja za hibridne mobilne aplikacije na ogrodju Cordova.
- Angular poskrbi za upravljanje drevesa komponent in nalaganje.
- Razvojna skupina Meteor (*MDG*) precej vlaga v izobraževanje za prehod in s tem nakazuje nejasno prihodnost Blaze.

3.8 Podpora React

Meteor omogoča tudi možnost uporabe *React*, ki je poleg Angular.js drugi predstavnik sodobnih ogrodij UI. Razvit je v sodelovanju družb Facebook in Instagram. Namenjen je reševanju enega problema: Gradnji kompleksnih aplikacij s podatki, ki se tekom časa spreminjajo [42].

React ni knjižnica *MVC*, ampak knjižnica JS namenjena implementaciji pogledov UI (*angl. views*) arhitekture MVC. Knjižnica je namenjena gradnji sestavljivih UI in spodbuja ustvarjanje večkrat uporabnih komponent UI (*angl. reusable UI components*).

React uporablja lastno predstavitev dokumenta. Ko se prva komponenta React vzpostavi, izvede metodo *render* in ustvari lažjo predstavitev UI. Iz te predstavitve je nato ustvarjeno označevanje nadbesedila (*angl. markup*), ki je vrinjeno v dokument. Ob vsaki naslednji spremembi podatkov se ponovno

kliče metoda *render*. Za zagotavljanje učinkovitosti osveževanja, se povratna vrednost predhodnega klica primerja z novo in ustvari minimalen nabor sprememb, ki se nato uporabijo na objektnem modelu dokumenta (*DOM*). Povratna vrednost ni niti niz, niti vozlišče DOM, ampak predstavlja opis, kakšen naj bo videz DOM.

React odlikujejo naslednje lastnosti:

- Enostavnost uporabe. Določimo le, kako naj bo aplikacija videti v določenem trenutku in React ob spremembi podatkov samodejno upravlja posodobitve UI.

- Je deklarativen. Ob spremembi podatkov, React konceptualno izvede osvežitev le-teh in poskrbi, da se posodobijo le spremenjeni deli.

- Omogoča gradnjo sestavljivih komponent. Ključni princip knjižnice React je vnovična uporaba komponent. V Reactu sestavljamo komponente, ki so vsebovane (*angl. encapsulated*) na način, da omogočajo vnovično uporabo kode, testov in ločitev vlog (*angl. separation of concern*).

React ne uporablja predlog HTML, kar je sicer običajno za gradnjo UI spletnih aplikacij. Je pravi programski jezik za izrisovanje pogledov in posameznih komponent UI. Prednosti takega pristopa pred predlogami HTML so:

- JS je fleksibilen in zmogljiv programski jezik, ki omogoča abstrakcije, kar je ključno za gradnjo kompleksnih aplikacij.

- Združeno označevanje nadbesedila (*angl. markup*) in pripadajoča logika omogočata enostavnejšo razširljivost in vzdrževanje.

- Manjša je ranljivost XSS, ker združitev označevanja nadbesedila in vsebine onemogočata veriženje nizov (*angl. string concatenation*).

JSX

V Reactu lahko opcijsko izberemo uporabo sintaktične razširitve *JSX*, ki nudi zapis, podoben sintaksi *XML* za definiranje drevesnih struktur z atributi. Zapis je priročen za vse, ki niso domači v JS, npr. za oblikovalce.

Poglejmo še izračuna ocen skupnosti za React (3.5) in JSX (3.6).

$$Ocena_{(facebook/react)} = 43170 + 7370 + 6821 + 715 \times 10 = 64511 \tag{3.5}$$

$$Ocena_{(facebook/jsx)} = 365 + 32 + 53 + 13 \times 10 = 580 \tag{3.6}$$

Ugotovimo, da ocena skupnosti (3.5) uvršča knjižnico React na drugo mesto med ogrodji UI, takoj za Angular.js (2.3). Nizka ocena skupnosti za sintaktične razširitve JSX nakazuje nizek sprejem.

3.9 Apache Cordova

Apache Cordova je ogrodje, ki nam omogoča gradnjo hibridnih mobilnih aplikacij. Cordova ovije aplikacijsko kodo HTML v domorodni kontejner, ki lahko dostopa do funkcionalnosti naprave na številnih platformah. Funkcionalnosti so dostopne preko enotnega JS APIja, kar omogoča uporabo iste kode na različnih platformah [43] .

Cordova je vključena v ogrodje Meteor, a je namestitev zaradi prihranka prostora opcijska, tako podporo za gradnjo mobilnih aplikacij omogočimo z ukazi:

- `meteor install-sdk android` in `meteor add-platform android`
- `meteor install-sdk ios` in `meteor add-platform ios`

Prvi ukaz prenese programska orodja za razvoj (SDK), drugi vključi izbrano platformo v našo aplikacijo Meteor. Aplikacijo Cordova nato gradimo in poganjamo z ukazom Meteor: `meteor run android-device`

3.9.1 Namestitev Apache Cordova

Apache Cordova namestimo z ukazom: `npm install -g cordova` . Novo aplikacijo ustvarimo z ukazom:

```
cordova create aplikacija si.fri.aplikacija Aplikacija
```

Ukaz pripravi ustrezno strukturo projekta naše aplikacije, kjer vstopno stran naše aplikacije predstavlja datoteka *www/index.html*. Strukturo projekta imamo tako pripravljeno vendar moramo v nadaljnjem koraku dodati še platforme, za katere želimo graditi aplikacije. To naredimo tako, da se prestavimo v mapo aplikacije in izvedemo enega ali več ukazov za dodajanje platforme:

```
cordova platform add ios --save
cordova platform add android --save
```

Za gradnjo mobilnih aplikacij je zahtevana tudi prisotnost ustreznih razvojnih orodij SDK, z ukazom `cordova requirements` lahko preverimo ali naš sistem vsebuje vse zahtevane razvojne komponente za gradnjo aplikacij za želeno platformo. Aplikacije lahko zgradimo za vse platforme naenkrat z ukazom `cordova build` , gradnjo pa lahko omejimo tudi na posamezno platformo, na primer z ukazom `cordova build ios` , prožimo le gradnjo za naprave iPad in iPhone.

Aplikacijo lahko preskušamo tudi v emulatorjih, ki so del SDK za posamezno platformo s pomočjo ukaza: `cordova emulate android` . Druga možnost je izvajanje neposredno na napravi, ki jo priključimo na razvojni računalnik z ukazom: `cordova run android`

Pri razvoju za Android je pomembno omeniti, da moramo na sistem Windows predhodno namestiti ustrezen gonilnik za vsako posamezno napravo Android. Te zahteve na sistemih Linux ali OS X ni, kar pomeni znaten prihranek časa.

3.9.2 Vtičniki Apache Cordova

Posamične funkcionalnosti mobilne naprave v aplikaciji omogočamo s pomočjo vtičnikov. Sistemski vtičniki Apache Cordova so predstavljeni v Tabeli 3.1,

dodatne najdemo s pomočjo iskalnika *Npm Plugin Search* [44]. Ta vtičnike prikaže kot kartice, ki vsebujejo vse ključne podatke, tudi podatek o platformah na kateri je posamezen vtičnik podprt.

Od aprila 2015 dalje so vtičniki Cordova vključeni v ekosistem Npm. Od ostalih paketov se ločijo po identifikatorju *cordova-plugin-**, za iskanje lahko uporabimo tudi spletno mesto `https://www.npmjs.com`.

Izbrane vtičnike, ki jih želimo uporabiti v svoji mobilni aplikaciji namestimo z ukazom: `cordova plugin add cordova-plugin-ime` Izračunajmo še oceno skupnosti z enačbo (3.7).

$$Ocena_{(apache/cordova-cli)} = 457 + 245 + 1143 + 73 \times 10 = 2575 \tag{3.7}$$

Ocena skupnosti je nizka, kar lahko pripišemo dejstvu, da se Cordova uporablja tudi in predvsem v drugih ogrodjih, kot sta Ionic (2.8) in Meteor (3.1).

3.10 Podatkovni sklad Apollo

Apollo je podatkovni sklad (*angl. data stack*) namenjen sodobnim aplikacijam zgrajen na osnovi *GraphQL*. Vsebuje ločen nabor paketov, odjemalca GraphQL in nabor orodij za strežnik GraphQL-JS ter različne odjemalske integracije. Trenutno je na voljo kot tehnični predogled in je zasnovan z namenom:

- možnosti postopne vpeljave v katero koli obstoječo aplikacijo JS,
- enostavne uporabe in
- zagotavljanja orodji, ki razvijalcem omogočajo vpogled in razumevanje, kaj se v aplikaciji dogaja.

3.10.1 GraphQL

GraphQL je tanka plast med odjemalcem in strežnikom, ki streže podatke odjemalcu. Določa skupen vmesnik med strežnikom in odjemalcem za povpra-

Tabela 3.1: Vtičniki Apache Cordova

opisno ime	vtičnik
Status akumulatorja	`cordova-plugin-battery-status`
Kamera	`cordova-plugin-camera`
Konzola	`cordova-plugin-console`
Kontakti	`cordova-plugin-contacts`
Naprava	`cordova-plugin-device`
Pospešek in gibanje	`cordova-plugin-device-motion`
Magnetni kompas naprave	`cordova-plugin-device-motion`
Pogovorna okna	`cordova-plugin-dialogs`
Branje in pisanje datotek	`cordova-plugin-file`
Prenos datotek	`cordova-plugin-file-trasfer`
Zemljepisna lokacija [45]	`cordova-plugin-geolocation`
Globalizacija	`cordova-plugin-globalization`
Brskalnik aplikacije	`cordova-plugin-inappbrowser`
Večpredstavnost	`cordova-plugin-media`
Večpredstavnostni zajem	`cordova-plugin-media-capture`
Omrežne informacije	`cordova-plugin-network-information`
Uvodni zaslon	`cordova-plugin-splashscreen`
Vibriranje [46]	`cordova-plugin-vibration`
Statusna vrstica	`cordova-plugin-statusbar`
Pravilnik dostopov	`cordova-plugin-whitelist`

ševanje in prevzem podatkov. V poizvedbi GraphQL odjemalec opiše podatke v želeni obliki in strežnik vrne JSON s podatki v zahtevani obliki. GraphQL je:

- Deklarativen. Format rezultatov določi odjemalec (Koda 3.4). Strežnik vrača natanko to, kar zahteva odjemalec (Koda 3.5).
- Kompozicijski, ker so poizvedbe hierarhičen nabor polj, oblikovan kot bodo oblikovani vrnjeni podatki. Kompozicija je naraven način opisovanja podatkov.
- Strogo tipiziran (*strong-typed*), kar zagotavlja veljavnost v okviru tipov podatkov GraphQL.

Poizvedni jezik in izvajalno okolje GraphQL sta razvita in v uporabi pri družbi Facebook od leta 2012. Referenčna implementacija graphql-js je knjižnica JS, ki predstavlja osnovo za celovite implementacije GraphQL in orodij. Poizvedba GraphQL je niz, ki ga strežnik interpretira in vrne podatke v formatu določenem v sami poizvedbi. To najbolje prikažemo na enostavnem primeru (Koda 3.4) in vrnjeni notaciji JSON (Koda 3.5).

```
{
  user(id: 1200304) {
    id,
    ime,
    jePrijatelj,
    slikaProfila(size: 50)  {
      uri,
      width,
      height
    }
  }
}
```

Koda 3.4: Poizvedba GraphQL

```
{
  "user" : {
```

```
      "id": 1200304,
      "ime": "Mojca Pokrajculja",
      "jePrijatelj": true,
      "slikaProfila": {
        "uri": "http://streznik.cdn/uporabnik/1200304.jpg",
        "width": 50,
        "height": 50
      }
    }
  }
```

Koda 3.5: Struktura rezultata odseva strukturo naše poizvedbe.

V enačbah (3.8), (3.9) in (3.10) bomo izračunali ocene skupnosti tehnologij podatkovnega sklada Apollo.

$$Ocena_{(apollostack/apollo-server)} = 36 + 5 + 17 + 3 \times 10 = 88 \tag{3.8}$$

$$Ocena_{(apollostack/apollo-client)} = 461 + 29 + 611 + 12 \times 10 = 1221 \tag{3.9}$$

$$Ocena_{(graphql/graphql-js)} = 3698 + 245 + 655 + 60 \times 10 = 5198 \tag{3.10}$$

Ker je skupina MDG prestavila podatkovni sklad Apollo v začetku leta 2016, sta nizki oceni strežnika (3.8) in odjemalca (3.9) Apollo pričakovani, saj gre za novo tehnologijo. Tudi ocena GraphQL (3.10) je razmeroma nizka, kar lahko pomeni nepoznavanje tehnologije in nizek sprejem med razvijalci.

3.11 Mantra

Mantra je nabor smernic za arhitekturo aplikacij Meteor [47]. Cilja arhitekture Mantra sta:

- Zagotovitev enostavnega vzdrževanja (*angl. maintainability*).
  Vzdrževanje je ključno za uspeh v večjih razvojnih skupinah, dosežemo

ga z modularnim preskušanjem (*angl. unit testing*) vseh delov aplikacije in definicijo standardov za vse sestavne dele.

- Pripravljenost na prihodnost (*angl. future proof*). JS nam omogoča neštete kombinacije tehnologij in modulov, ki omogočajo več kot eno optimalno pot do rešitve problema. Težko napovemo, kaj je trenutno najboljše in kaj se bo v prihodnosti spremenilo. Mantra se zanaša na ključne principe za katere lahko predvidevamo, da bodo veljali dlje časa in dopušča možnost sprememb po potrebi.

Mantra posveča precej pozornosti odjemalskemu delu aplikacije. Kode odjemalca in strežnika ne meša in priporoča skupno rabo kode:

- Odjemalec zahteva veliko truda in predstavlja večji del kode. Upravljanje strežniške kode je običajno lažje obvladljivo.

- V prihodnosti bo odjemalski del aplikacije povezan s strežniškim delom na osnovi sheme. Odjemalcu ni treba vedeti, kakšna je implementacija strežniškega dela.

- Mantra ne verjame v univerzalne aplikacije. Spodbuja implementacijo ločenih aplikacij za različne platforme ob skupni rabi kode. Običajno je, da več odjemalskih aplikacij deluje v navezavi z enim strežnikom.

React je uporabljen za uporabniški vmesnik (UI). Mantra zahteva uporabo komponent, ki jih uvozimo iz Npm ali definiramo v sami aplikaciji, ter uporabo kontejnerjev.

Poslovno logiko definiramo v okviru akcij (*angl. actions*) in obsega:

- validacije,

- upravljanje stanja (*angl. state management*) in

- interakcije z oddaljenimi viri podatkov.

Akcija je enostavna funkcija, ki kot prvi argument sprejme celoten kontekst aplikacije. Vse kar izvajamo v okviru akcije je na osnovi podanega konteksta in dodatnih argumentov.

Aplikacija upravlja z različnimi stanji, ki jih lahko razdelimo na lokalno stanje in oddaljeno stanje. Na področju upravljanja stanja potekajo številne inovacije, zato je Mantra fleksibilna pri izbiri rešitve:

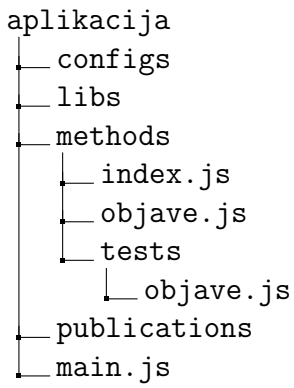
- *Meteor/MiniMongo* (oddaljeno stanje)
- *Tracker/ReactiveDict* (lokalno stanje)
- *FlowRouter* (lokalno stanje)
- *Redux* (lokalno stanje)
- *GraphQL* (oddaljeno stanje)
- *Falcor* (oddaljeno stanje)

Navkljub fleksibilni izbiri, Mantra zahteva nekaj pravil pri upravljanju stanj:

- Operacije s pisanjem se morajo izvesti v akciji.
- Branja se lahko izvajajo v akcijah in kontejnerjih.
- Branj in pisanj ne smemo izvajati neposredno v komponente UI. Komponente UI ne smejo vedeti ničesar o stanju aplikacije.

Kontejnerji so integracijska plast v Mantri in izvajajo:

- Uporabo stanja za spremembo spremenljivk in posredovanje le-teh v UI preko lastnosti.
- Posredovanje akcij v komponente UI.
- Posredovanje elementov konteksta aplikacije v komponente UI.



Slika 3.3: Priporočena struktura Mantra

Kontejnerji so komponente React in jih sestavljamo z uporabo *react-komposer*. Omogočena je uporaba različnih podatkovnih virov, vključno z *Meteor/-Tracker*, *Promises*, *Rx.js Observable*.

Za strežniški del aplikacije Mantra priporoča imeniško strukturo predstavljeno na Sliki 3.3.

Podrobneje si poglejmo še namen posameznih map.

- **configs** ... Mapa *configs* je mesto za nastavitve aplikacije. Nastavitve naj imajo privzeto izvozno funkcijo, koda konfiguracije naj ostane v funkciji sami. Primer datoteke *config.js* je prikazan v Kodi 3.6.

- **libs** ... Mapa *libs* je namenjena knjižnicam pomožnih funkcij, ki jih želimo uporabiti na strežniškem delu aplikacije.

- **methods** ... Mapa *methods* vsebuje metode za posamezne funkcionalnosti, datoteke poimenujemo z imenom vsebovane metode. Datoteka *index.js* uvaža vse ostale module v imeniku in jih izvaža, na ta način lahko uvozimo vse metode naenkrat.

- **tests** ... Mapa *tests* vsebuje metode za modularno preskušanje (*angl. unit testing*).
- **publications** ... Mapa *publications* je identična mapi *methods* a je namenjena objavam (*angl. publications*).
- **main.js** ... Datoteka *main.js* je vstopna točka v aplikacijo, kot vidimo v primeru Kode 3.7. Tu uvozimo naše metode, objave (*angl. publications*) in nastavitve.

```
1 export default function() {
2   // tu kličemo konfiguracije
3 }
```

Koda 3.6: Primer datoteke *config.js* za nastavitve aplikacije.

```
1 import publications from './publications';
2 import methods from './methods';
3 import addInitialData from './configs/initial_adds.js';
4
5 publications();
6 methods();
7 addInitialData();
```

Koda 3.7: Uvažanje metod, objav in nastavitev v datoteki *main.js*.

Poglavje 4

Dodatne tehnologije za celostni razvoj

Pogledali si bomo še nekaj tehnologij, ki temeljijo na JS in Node.js in niso zajete niti v skladu tehnologij MEAN niti ogrodju Meteor, a so lahko del celostnega razvoja sodobnih aplikacij:

- tehnologije za gradnjo namiznih aplikacij,
- tehnologije za storitve v oblaku in uporaba Node.js v mikroservisih in
- tehnologije za aplikacije interneta stvari (*angl. internet of things, IoT*).

4.1 Tehnologije za namizne aplikacije

Podobno razvoju mobilnih aplikacij, tudi razvoj namiznih (*angl. desktop*) aplikacij običajno zahteva poznavanje dodatnega razvojnega okolja. Običajno so to *Visual Studio* za razvoj aplikacij za operacijske sisteme Microsoft Windows in *Xcode* za razvoj aplikacij za operacijski sistem OS X proizvajalca Apple, in različna orodja za razvoj Linux aplikacij, kot so *GTK+*, *Glade* ali *Anjuta*.

Razvoju na navzkrižnih platformah (*angl. cross-platform development*) je doslej služila predvsem Java, a se navkljub odličnim razvojnim okoljem

IDE, kot so *Eclipse* in njegove izpeljanke, sodeč po številčnosti aplikacij ni najbolj uveljavila.

Nasprotno se je v zadnjem času na področju aplikacij razvitih v JS poraja pestra paleta aplikacij, za katere uporabniki večinoma niti ne opazimo, da se za namizno aplikacijo skrivata Node.js in Chromium oz. koda JS.

Za razvoj in gradnjo namiznih aplikaciji iz kode JS imamo na voljo vsaj tri tehnologije:

- *NW.js*, ki dobesedno pomeni Node.js - Webkit,
- Electron in
- CEF (*Chronium Embedded Framework*), pri katerem se aplikacija gradi v programskem jeziku C.

Tehnologije se razlikujejo v naslednjih točkah:

- sistemu gradnje,
- vstopni točki v aplikacijo,
- načinu integracije Node.js in
- kontekstualnosti.

NW.js

Pri NW.js je vstopna točka spletna stran, katere URL določimo v datoteki *package.json*, ta se nato odpre v oknu brskalnika, kot glavno okno aplikacije. Integracija Node.js v Chronium je izvedena s pomočjo popravkov kode Chronium. NW.js zaradi načina implementacije ločuje med kontekstom Node.js in spletnim kontekstom.

Electron

Electron deluje na operacijskih sistemih Mac OS X, Windows in Linux. Namestitev izvedemo z ukazom: `npm install -g electron-prebuilt` .

Struktura aplikacije Electron (Slika 4.1) je lahko zelo enostavna. Vstopna točka v aplikacijo je datoteka JS, ki jo določimo v datoteki *package.json*. Kot vidimo v Kodi 4.1, je ta strukturno enaka, kot smo je vajeni iz modulov Node.js. Vsebino datoteke HTML naložimo preko APIja v datoteki JS določeni v atributu *main*, kjer se tudi naročimo na okenske dogodke, ter se tako določimo kdaj zapustiti aplikacijo.

Datoteka *main.js* mora poskrbeti za ustvarjanje oken in upravljanje sistemskih dogodkov. Datoteka *index.html* je spletna stran, ki jo prikažemo v našem glavnem oknu namizne aplikacije.

Delovanje Electrona je bližje delovanju Node.js saj so APIji nizkonivojski. Electron uporablja skupno knjižnico *libchroniumcontent* za dostop do Chronium Content APIjev, zato ne zahteva zmogljivega računalnika za gradnjo aplikacije. Electron s pomočjo kode *node_bindigns* integrira zanko *libuv* s sporočilno zanko posamezne platforme brez posegov v kodo Chromium. Ne uvaja novega spletnega konteksta, ker uporablja več-kontekstualnost (*angl. multicontextuality*), ki je funkcionalnost okolja Node.js.

Electron med drugim omogoča tudi samodejne posodobitve, poročila o sesutjih, namestitvene programe za Windows, razhroščevanje in profiliranje, sistemske menije in obvestila.

```
1 {
2   'name': 'aplikacija',
3   'version': '0.0.1',
4   'main': 'main.js'
5 }
```

Koda 4.1: Datoteka *package.json* aplikacije Electron.

```
aplikacija
├── package.json
├── main.js
└── index.html
```

Slika 4.1: Struktura aplikacije Electron.

4.1.1 CEF

CEF predstavlja možnost vgradnje podporne infrastrukture v lastne aplikacije, ki omogoča uporabo izrisa HTML in podporo JS. Aplikacije osnovane na distribuciji binarne kode CEF se zato gradijo s pomočjo standardnih orodij za posamezne operacijske sisteme:

- Visual Studio na sistemu Windows
- Xcode na sistemu Mac OS X
- gcc/make na sistemu Linux

Gradnja aplikacij se tako razlikuje glede na izbrano platformo. Na voljo so tudi navezave (*angl. bindings*) za programske jezike Delphi, Go, Java, C# in Python. Skupne korake lahko povzamemo:

- Prevajanje statične knjižnice *libcef_dll_wrapper*
- Prevajanje izvorne kode aplikacije in povezovanje z dinamično knjižnico *libcef* in statično knjižnico *libcef_dll_wrapper*
- Kopiranje knjižnic in virov (datotek *.html*, *.js*, *.css*, nizov bededil, itd.) v izhodno mapo.

Poglejmo si še ocene skupnosti tehnologij za gradnjo namiznih aplikacij izračunanih v enačbah (4.1) in (4.2).

$$Ocena_{(nwjs/nw.js)} = 28567 + 3203 + 2547 + 69 \times 10 = 35007 \quad (4.1)$$

$$Ocena_{(electron/electron)} = 31432 + 3318 + 9332 + 392 \times 10 = 48002 \quad (4.2)$$

Ugotovimo, da ima ogrodje Electron boljšo oceno skupnosti. Ocene skupnosti za CEF nismo uspeli pridobiti, ker ni na voljo na shrambi kode GitHub. Neglede na to, je CEF uporabljen pri gradnji številnih priljubljenih aplikacij [48].

4.2 Tehnologije za storitve v oblaku

Vsi ponudniki storitev v oblaku vključujejo tudi podporo okolju Node.js, običajno je JS med bolje podprtimi programskimi jeziki. Podpora ostalim programskih jezikom se precej bolj razlikuje med posameznimi ponudniki storitev v oblaku. Node je danes ena hitreje rastočih platform skoraj pri vseh naštetih ponudnikih storitev v oblaku.

4.2.1 AWS Lambda

Za pripravo lastnih paketov Node.js za storitev *AWS Lambda*, potrebujemo instanco *Amazon EC2* ali sistem z nameščenim operacijskim sistemom *Amazon Linux* in Node.js. Za pripravo paketa za AWS Lambda lahko uporabimo obstoječe module Node.js, ki jih poiščemo na Npm. Najprej ustvarimo mapo, ki bo vsebovala našo funkcijo Lambda in vse njene module.

Namestimo Npm paket AWS SDK in preverimo namestitev s pomočjo testne funkcije, ki izpiše različico, kot prikazano v Kodi 4.2.

```
1 npm install --prefix=~/lambdaTestFunction aws-sdk
2 echo 'var AWS = require("aws-sdk");console.log(AWS.EC2.
     apiVersions)'> test.js
3 node test.js
```

Koda 4.2: Nameščanje AWS SDK.

Če želimo uporabljati npr. knjižnico OpenCV, namestimo razvojna orodja, kot je prikazano v Kodi 4.3.

```
sudo yum update
sudo yum install gcc44 gcc-c++ libgcc44 cmake -y
wget http://nodejs.org/dist/v0.10.33/node-v0.10.33.tar.gz
tar -zxvf node-v0.10.33.tar.gz
cd node-v0.10.33 && ./configure && make
sudo make install
```

Koda 4.3: Nameščanje razvojnih orodij.

Zatem lahko prenesemo izvorno kodo OpenCV in knjižnico zgradimo, kot je prikazano v Kodi 4.4.

```
wget http://softlayer-dal.dl.sourceforge.net/project/
   opencvlibrary/opencv-unix/2.4.9/opencv-2.4.9.zip
mkdir opencv_install
mkdir opencv_example
unzip opencv-2.4.9.zip -d opencv_install/ && cd
   opencv_install
cmake -D CMAKE_BUILD_TYPE=RELEASE -D BUILD_SHARED_LIBS=NO -D
   CMAKE_INSTALL_PREFIX=~/opencv opencv-2.4.9/
make && make install
```

Koda 4.4: Prenos in gradnja OpenCV.

Ko je knjižnica zgrajena, lahko namestimo Npm modul opencv, kot je prikazano v Kodi 4.5. Predpono poti PKG_CONFIG_PATH določimo preden prožimo `npm install` , da vključimo nove statične knjižnice v lokalno pot.

```
PKG_CONFIG_PATH=~/opencv/lib/pkgconfig/
npm install -prefix=~/opencv_example opencv
```

Koda 4.5: Nameščanje modula Node.js.

Node.js modul za OpenCV vsebuje nekaj testnih funkcij za zaznavanje obrazov. Z njimi lahko, kot je prikazano v Kodi 4.6, preverimo ali je modul pravilo zgrajen. Če je temu tako, bo v izhodni datoteki s krogom označen obraz na sliki Mona Lize.

```
cd ~/opencv_example
cd node_modules/opencv/examples/
mkdir tmp
node face-detection.js
Image saved to ./tmp/face-detection.png
```

Koda 4.6: Testiranje delovanja OpenCV v Node.js.

Preostane nam le, da prilagodimo svojo lambda funkcijo, odstranimo testno kodo, mapo zgostimo v datoteko ZIP in jo namestimo v oblak.

4.3 Razvoj aplikacij za internet stvari

Razvoj aplikacij za internet stvari (*angl. internet of things, IoT*) je novo, zanimivo in kompleksno področje. Za programiranje številnih novih naprav IoT so se morali doslej razvijalci posluževati precej prilagojenih programskih jezikov in namenskih razvojnih okolji IDE. Proizvajalci priljubljenih naprav namenjenih prototipnemu razvoju IoT so kmalu ugotovili, da je Node.js zelo primeren tudi za naprave IoT iz več razlogov:

- Prinaša dvajset let izkušenj dogodkovnega programiranja JS.
- Dogodki so naraven način delovanja IoT naprav: dogodek proži neko akcijo.
- Node omogoča dostop do IoT tehnologij široki skupini razvijalcev.
- Omogočena je uporaba obstoječih modulov JS.

V zadnjih dveh letih je tako prišlo do pomembnega preobrata, saj so večji proizvajalci strojnih platform IoT začeli Node.js vključevati na svoje naprave. Tudi na naprave z zelo nizko porabo energije, kot so npr. *Intel Galileo*, *Intel Edison* in *Tessel 2*. Izkaže se, da je Node.js zelo primerno okolje, ker ni intenziven porabnik virov in lahko opravi tudi ogromno operacij V/I ob zelo majhni bremenitvi pomnilnika ali CPE.

Anketa fundacije Node.js je v januarju 2016 pokazala, da 96% sodelujočih na vprašanja vezana o IoT, razvija s pomočjo JS oz. Node.js [49]. Že skoraj 10% anketirancev uporablja Node.js na odjemalcu, zalednih sistemih in za razvoj IoT.

4.3.1 Node-RED

Node-RED je grafično orodje za povezovanje IoT [51] zgrajeno na Node.js in v celoti izkorišča prednosti, ki jih prinaša dogodkovni model programiranja. Grafični uporabniški vmesnik (GUI) za manipulacijo tokov, prikazan na Sliki 4.2, omogoča povezovanje strojnih naprav, APIjev in spletnih servisov. Funkcije JS lahko ustvarimo s pomočjo urejevalnika v samem okolju.
Omogočeno je shranjevanje funkcij, predlog in vnovična uporaba predhodno ustvarjenih tokov. Omogoča uporabo obstoječih modulov NPM in s tem enostavno razširljivost, kot tudi deljenje tokov, ustvarjenih z Node-RED v zapisu JSON, kar omogoča enostavno uvažanje in izvažanje. Node-RED lahko izvajamo na nizkocenovni strojni opremi, kot je Raspbery Pi, kot tudi v oblaku. Node-RED namestimo z ukazom: `sudo npm install -g node-red` in ga poženemo z ukazom: `node-red` Ob namestitvi že vsebuje nekaj osnovnih vozlišč, dodatna vozlišča ali tokove lahko poiščemo v knjižnici Node-RED ali repozitoriju Npm. Node-RED lahko tudi vgradimo v lastno aplikacijo. Običajen scenarij je uporaba Node-RED za generiranje podatkovnih tokov, ki jih želimo prikazati na pregledni plošči.

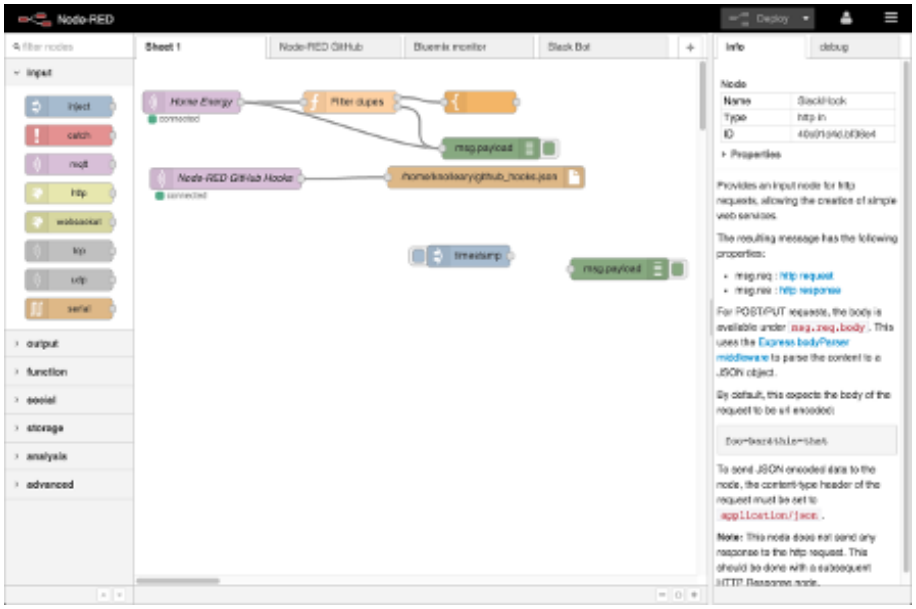
V enačbi (4.3) si oglejmo še oceno skupnosti.

$$Ocena_{(node-red/node-red)} = 2690 + 678 + 2006 + 31 \times 10 = 5684 \qquad (4.3)$$

Na podlagi razmeroma nizke ocene (4.3) lahko sklepamo, da gre za relativno novo tehnologijo.

4.3.2 Ogrodje IoT.js

IoT.js je ogrodje za gradnjo IoT na osnovi JS interpreterja *JerryScript* in knjižnice *libuv* za asinhroni V/I. JerryScript je JS-stroj namenjen izvajanju



Slika 4.2: Grafično okolje Node-RED omogoča povezovanje naprav IoT in storitev.

na mikrokrmilnikih z omejenimi viri: nekaj kilobajti pomnilnika (RAM) (<64 KB) in omejeno količino trajnega pomnilnika (ROM) (<200 KB) namenjenega interpreterju JS. Interpreter JerryScript omogoča prevajanje na napravi, izvajanje in dostop do perifernih enot iz kode JS. Poglejmo si ocene skupnosti v enačbah (4.4), (4.5) in (4.6).

$$Ocena_{(Samsung/iotjs)} = 939 + 129 + 244 + 15 \times 10 = 1462 \tag{4.4}$$

$$Ocena_{(Samsung/jerryscript)} = 1318 + 151 + 2004 + 33 \times 10 = 3803 \tag{4.5}$$

$$Ocena_{(libuv/libuv)} = 4135 + 688 + 3538 + 228 \times 10 = 10641 \tag{4.6}$$

Vidimo lahko, da je knjižnica za asinhroni V/I najbolje ocenjena, kar lahko pripišemo širši možnosti uporabe.

Poglavje 5

Izbira sklada tehnologij za izbran praktični primer

Zahteve, ki jih bomo skušali rešiti na našem praktičnem primeru so štiri:

1. Zajem statične slike na različnih napravah
2. Obdelava zajete slike
3. Knjižnice za računalniški vid
4. Povezljivost s spletnimi servisi

5.1 Zajem slike na napravah

Za izpolnitev prve zahteve sklad tehnologij MEAN ni zadosten, saj v trenutnem stanju še ne vsebuje zadovoljive podpore za zajem slike. Izboljšave standarda *HTML5* in podpora *WebRTC* sicer napovedujejo tudi možnost uporabe kamere v poljubnem sodobnem brskalniku, kar pa še ne pomeni, da zajem slike deluje tudi na vsaki napravi. Poleg tega so pričakovanja uporabnikov tudi domorodne aplikacije za njihovo mobilno napravo. Glede na trenutni tržni delež, moramo poskrbeti najmanj za zagotovitev aplikacije za mobilne naprave Android in Apple iOS. Dobrodošla bi bila tudi podpora na namiznih računalnikih.

Tabela 5.1: Ocena ustreznosti tehnologij

	spletni zajem slike	zajem slike Android	zajem slike iOS	namizna aplikacija	enostavnost implementacije	skupaj	odprtokodnost
MEAN	4	-	-	-	5	9	18
Meteor	5	5	5	-	5	20	**40**
Apache Cordova	-	5	5	-	5	15	30
Electron	-	-	-	5	5	10	20

Za sklad tehnologij MEAN, lahko hitro ugotovimo, da temu ne ustreza. Kot vidimo v Tabeli 5.1, dodatno potrebujemo še najmanj tehnologijo Apache Cordova, ki zagotavlja podporo kameri na napravi. Pri skladu vključenem v Meteor je Apache Cordova že vsebovana tehnologija. Dodatno bi morda potrebovali še tehnologijo Electron, če bi uporabniku želeli zagotoviti tudi namizno aplikacijo. Tudi čas implementacije osnovnega zajema slike je z ogrodjem Meteor krajši.

5.1.1 Zajem slike v ogrodju Meteor

Ogrodje meteor omogoča enostaven zajem slik iz kamere pametnega telefona ali namiznega računalnika s pomočjo paketa *Meteor Camera Package*. Za uporabo paketa ga najprej dodamo z ukazom: `meteor add mdg:camera` in za zajem slike ustvarimo dodatne predlogo HTML, ki vsebujejo elemente s pomočjo katerih bomo rokovali zajem slike ter ustrezne funkcije JS, kot je prikazano v primerih Kode 5.1 in 5.2.

```
<head>
  <title>Moj fotoaparat</title>
</head>
<body>
  <h1>Moj fotoaparat</h1>
  {{> camera_button}}
  <div id="picture">
    <img src={{picture}}>
  </div>
</body>

<template name="camera_button">
   <button id="camera-button">Zajem slike!</button>
</template>
```

Koda 5.1: *client/main.html* ... Predloga Blaze za UI fotoaparata.

```
Template.camera_button.events({
  'click #camera-button': function(){
    // Nastavitve zajema slike
    var options = { width: 640, height: 640, quality: 100 };
    // Klic zajema slike
    MeteorCamera.getPicture(options, function(error, data){
      if (error){ // Rokovanje napak
        console.log(error);
      }else{  // Sliko shranimo kot base64-kodiran podatkovni URI
        Session.set('zajetaSlika', data);
      }
    });
  }
});
// pomožna funkcijo, ki bo skrbela za prikaz zajete slike
Template.body.helpers({
  picture: function(){
    return Session.get('zajetaSlika');
  }
});
```

Koda 5.2: *client/fotoaparat.js* ... Zajem in prikaz slike.

Poglejmo oceno skupnosti modula za zajem slike v ogrodju Meteor podano z enačbo (5.1).

$$Ocena_{(meteor/mobile-package)} = 324 + 144 + 74 + 12 \times 10 = 662 \quad (5.1)$$

5.1.2 Zajem slike s pomočjo Apache Cordova

Podporo posameznim funkcionalnostim Apache Cordova omogoča s pomočjo vtičnikov. Funkcionalnosti kamere naprave so nam na voljo preko namenskega vtičnika cordova-plugin-camera, ki ga namestimo z ukazom: `cordova plugin add cordova-plugin-camera` . Vtičnik definira globalni objekt `navigator.camera` preko katerega so nam na voljo klici API za zajem slik. Objekt `navigator.camera` je na voljo šele po dogodku `deviceready` . V izseku Kode 5.3 vidimo implementacijo zajema slike.

```
1  navigator.camera.getPicture(onSuccess, onFail, {
2      quality: 50,
3      destinationType: Camera.DestinationType.DATA_URL
4  });
5  function onSuccess(imageData) {
6      var image = document.getElementById('slika');
7      image.src = "" + imageData;
8  }
9  function onFail(message) {
10     alert('Zajem slike ni uspel: ' + message);
11 }
```

Koda 5.3: *main.js* ... Zajem slike z vtičnikom Apache Cordova.

Poglejmo oceno skupnosti vtičnika Cordova za zajem slike izračunano v enačbi (5.2).

$$Ocena_{(apache/cordova-plugin-camera)} = 379 + 518 + 439 + 66 \times 10 = 1996 \quad (5.2)$$

Vidimo lahko, da ima vtičnik Cordova (5.2) višjo oceno skupnosti od modula Meteor (5.1).

5.1.3 Zajem slike s pomočjo WebRTC

Na kratko poglejmo možnost zajema slike s pomočjo *WebRTC*. Ta je najbolj primeren za zajem in obdelavo videa. *WebRTC* je zamišljen kot nabor vmesnikov API, ki omogočajo komunikacijo v realnem času (*angl. Real Time Comunication, RTC*) v brskalnikih in mobilnih aplikacijah.

Del WebRTC APIjev je tudi metoda *GetUserMedia*, ki omogoča zajem zvoka in videa. Kot primer si bomo pogledali knjižnico *clmtrackr* [58], ki omogoča nekoliko naprednejše zaznavanje obraza v JS (Slika 5.1). Ker knjižnica uporablja tudi *WebGL* je njena uporaba primerna predvsem na računalnikih.

Primer aplikacije Meteor, ki uporablja WebRTC najdemo na repozitoriju: `https://github.com/foxdog-studios/meteor-webrtc`.

Ker je količina prenesenih podatkov v takem primeru nekajkrat zajetnejša, je smiselna uporaba le za primere uporabe, kjer predhodna obdelava na odjemalski napravi ne pride v poštev. Veliko bolj smiselna je realnočasovna obdelava v oblaku na zahtevo ob izpolnjenih pogojih. Primer si lahko predstavljamo spletno kamero, ki spremlja določen kader. Že na sami napravi preverjamo prisotnost gibanja, ob prisotnosti gibanja lahko na zajeti sliki preverimo na primer prisotnost obrazov. V primeru, ko so zadoščeni v naprej določeni pogoji, tok videa začnemo posredovati video-analitičnem mikroservisu v oblaku. Posredovanje video toka se zaključi, ko video-analitični sistem javi odjemalcu ustrezen ukaz.

5.2 Obdelava in analiza zajete slike

Osnovna ideja izomorfne (*angl. isomorphic*) kode JS je v tem, da lahko isto kodo izvajamo tako na odjemalcu, kot na strežniku. Za določene operacije obdelave slike lahko tako izberemo knjižnico JS in jo bodisi uporabimo na spletni strani oz. v mobilni aplikaciji ali na zalednem strežniškem sistemu, kjer se ista knjižnica izvaja v Node.js.

Za naš primer bomo poiskali knjižnico, ki omogoča preprosto detekcijo

obraza na zajeti sliki. V iskalniku na spletni strani `www.npmjs.com` vtipkamo ključni besedi *"face detection"* in dobimo več kot 35 rezultatov.

Za podrobno analizo vseh knjižnic bi se morali nekoliko bolje poglobiti, ampak že ob hitrem pregledu lahko izločimo nekaj knjižnic, ki ustrezajo našim potrebam.

Ena bolje ocenjenih je *opencv* [54]. Opis nam pove, da gre za modul Node.js, ki nam omogoča navezavo s priljubljenim odprtokodnim sistemom za računalniški vid *OpenCV* [55]. Ker ta knjižnica zahteva tudi vključevanje zunanjih knjižnic, ki niso v JavaScript-u, ne omogoča izomorfne kode. Knjižnico seveda lahko, kljub temu uporabimo za programiranje spletnega strežnika oz. vmesnikov API za storitve računalniškega vida v JS. Programsko opremo *OpenCV* v tem primeru namestimo na naš strežnik.

Najdemo povsem osnovni modul *face-detect* [60] v čistem JavaScriptu. Kot tudi zmogljivejše module JS, kot je npr. *clmtrackr* [58]. Ta omogoča prileganje modelov obrazu in zaznavanje obraza na videu (Slika 5.1) ob predpostavki, da imamo v brskalniku podporo za WebGL, ki se uporablja za pospeševanje. Delovanje modula *clmtrackr* smo uspešno preverili tudi v namizni aplikaciji, ki smo jo pripravili v ogrodju Electron.

Med knjižnicami najdemo tudi nekaj takih, ki predstavljajo podporne knjižnice za navezavo na mikro storitve v oblaku za analizo slik. Takšen primer je Node.js modul *project-oxford*, ki ponuja implementacijo vmesnikov za *Project-Oxford API* [57] in omogoča zaznavo enega ali več obrazov, identifikacijo obrazov, ujemanje obrazov, analizo čustev obraza, analizo slike in OCR. Ustreznost knjižnic, ki omogočajo zaznavo obraza v Node.js, smo ocenili v Tabeli (5.2).

5.3 Nalaganje slik na spletni strežnik

5.3.1 Nalaganje slik s pomočjo DDP

Za nalaganje slik na strežnih imamo na voljo več različnih možnosti. Glede na to, da ima Meteor na voljo podatkovno bazo Minimongo na odjemalcu in

Slika 5.1: Primer uporabe knjižnice *clmtrackr* [58].

Tabela 5.2: Knjižnice Npm za metode računalniškega vida.

	izomorfna koda	zaznava obraza	zaznava več obrazov	analiza čustev	analiza slike	identifikacija oseb	skupaj	odprtokodnost
opencv [55]	-	5	5	4	4	2	16	**32**
clmtrackr [58]	5	5	-	5	-	-	15	30
jsfeat [59]	5	5	-	-	-	-	10	20
face-detect [60]	5	5	-	-	-	-	10	20
tracking [61]	5	5	5	-	-	-	15	30
project-oxford [57]	5	5	5	5	5	5	30	30

MongoDB na strežniku, ki sta povezana prek protokola DDP, lahko zajete slike vstavimo neposredno v zbirko dokumentov na odjemalcu, ki se nato sinhronizira na strežnik in po potrebi na druge odjemalce, ki so naročeni na prejemanje posodobitev zbirke.

5.3.2 Nalaganje slik po protokolu HTTP

Druga možnost je nalaganje na spletni strežnik Meteor po protokolu HTTP, kjer metapodatke hranimo v MongoDB ločeno.

5.3.3 Nalaganje slik v spletno shrambo v oblaku

Tretja možnost je uporaba ločene shrambe za slike in shranjevanje metapodatkov v lastno podatkovno bazo MongoDB. Primer take implementacije je meteor knjižnica *Slingshot*, dostopna na naslovu: `https://github.com/CulturalMe/meteor-slingshot/`.

Implementacija s pomočjo zunanjih spletnih shramb prinaša številne prednosti, predvsem je ta pristop primeren, ko želimo uporabiti tudi storitve računalniškega vida v oblaku, saj številni ponudniki tovrstnih storitev zahtevajo posredovanje slik v njihove spletne shrambe. Pristop velja posnemati tudi v primeru implementacije lastne shrambe za fotografije.

Poglavje 6

Razvoj aplikacije

6.1 Priprava razvojnega okolja

6.1.1 Izbira računalnika

Za celosten odjemalsko-strežniški razvoj bomo izbrali računalnik sodobne izdelave, primernih procesorskih zmogljivosti in s primerno količino pomnilnika in diskovnega prostora. Upoštevati moramo namreč, da računalnik ni namenjen zgolj pisanju dopisov in obračanju razpredelnic. Tekom razvoja se vse bolj pojavlja tudi potreba po virtualizaciji operacijskih sistemov in izvajanju celotnih skladov tehnologij na enem ali več virtualnih strojih. Alternativa je lahko prosta dostopnost razvojnih strojev v oblaku.

6.1.2 Izbira operacijskega sistema

Kriteriji za operacijskega sistema so naslednji:

1. ukazna vrstica in podpora standardnim ukazom UNIX,
2. podpora verigam orodij za razvoj mobilnih aplikacij,
3. priklopa mobilnih naprav brez nameščanja dodatnih gonilnikov za vsako posamično napravo,

Tabela 6.1: Primernost operacijskih sistemov

	UNIX CLI	razvojna orodja	gonilniki	brskalniki	skupaj	odprtokodnost
Linux	5	2	3	3	13	26
OS X	5	4	5	3	17	**34**
Windows	1	4	1	3	9	9

4. možnost testiranja spletnih aplikacij na vseh vodilnih brskalnikih.

Ocene ustreznosti naštetim kriterijem lahko vidimo v Tabeli 6.1

Vidimo, da operacijski sistem OS X zadostuje skoraj vsem naštetim zahtevam z izjemo zadnje točke, ki je premostljiva s pomočjo virtualizacije. OS X je sistem UNIX s podporo ukazni vrstici Bash in vključuje standardne ukaze za povezovanje z oddaljenimi strežniki, kot so npr. *ssh*. Številne tehnologije sistema OS X, vključno z jedrom UNIX, so na voljo tudi kot odprta koda [52].

Izbiri sistema pripomore tudi to, da želimo pripraviti tudi mobilno aplikacijo za naprave *iPad* in *iPhone*. Za gradnjo in testiranje aplikacij *iOS* tako potrebujemo tudi orodje *Xcode*, ki sicer ni na voljo na drugih operacijskih sistemi.

Ostane nam nepokrita možnost razvoja za *Windows Phone*, ki je zanemarljiva zaradi nizkega tržnega deleža [35] in jo lahko premostimo z namestitvijo dodatnega sistema s programom *BootCamp* [53].

6.1.3 Izbira in namestitev urejevalnika

Zrelost programskega jezika JS dokazujejo tudi številni namizni programi razviti v JS. Za naš razvoj bomo izbrali enega izmed priljubljenih urejevalnikov kode oz. interaktivno razvojno okolje (*angl. Interactive Development*

Environment, IDE), ki je razvit v JS in dostopen na vseh ključnih namiznih operacijskih sistemih. Vsi našteti urejevalniki omogočajo udobno urejanje kode JS in HTML, delo s priljubljenimi distribuiranimi sistemi upravljanja kode in razširljivost. Seveda je tudi razvoj razširitev in vtičnikov IDE v programskem jeziku JS.

Atom

Prva možnost na tem področju je odprtokodni urejevalnik *Atom*. Atom je odprtokodna namizna aplikacija zgrajena s pomočjo HTML, JS, CSS z integracijo Node.js, ki se izvaja se na ogrodju Electron. Urejevalnik že presega milijon aktivnih uporabnikov in ima bogat ekosistem vtičnikov.

Visual Studio Code

Alternativno lahko uporabimo tudi urejevalnik *Visual Studio Code*, ki je prvi urejevalnik programske kode in prvo razvojno orodje družine Microsoft Visual Studio, ki je na voljo na sistemih OS X, Linux in Windows. Če izberemo slednjega in si želimo olajšati dostop do programa iz ukazne vrstice, po namestitvi poženemo Visual Studio Code, odpremo ukazno paleto (F1) in vpišemo *"shell command"* in izvedemo *"Install 'code' command in PATH command"*, zatem vnovič poženemo terminal z ukazno vrstico Bash, kjer bo Visual Studio Code dostopen z ukazom *"code"* in bo omogočeno urejanje datotek v poljubni mapi s priročnim ukazom `code .` .

Nuclide

Tretja alternativa, ki prav tako temelji na urejevalniku Atom je nastala pod okriljem odprtokodnih projektov družbe Facebook, je *Nuclide*. Nuclide bo s časom verjetno prva izbira za vse razvijalce, ki bodo za celosten razvoj odjemalsko-strežniških aplikacij uporabljali Facebookovo ogrodje React oz. React Native. Pomembna razlika je, da Nuclide za razliko od prejšnjih dveh urejevalnikov privzeto nudi boljšo podporo distribuiranemu sistemu upravl-

janja izvorne kode Merculial in je tako boljša izbira za vse, ki v svojem razvojnem procesu uporabljajo predvsem Mercurial.

Brackets

Brackets je četrti IDE in je plod razvoja podjetja Adobe. Njegova prednost je predvsem boljša integracija z orodji in produkti družbe Adobe. V mislih imamo predvsem storitve, kot je npr. Creative Cloud Extract, ki omogoča ekstrakcijo oblikovalskih podatkov, kot so barve, pisave, CSS in slike, iz datotek PSD.

V enačbah (6.1), (6.2), (6.3) in (6.4) si poglejmo ocene skupnosti naštetih urejevalnikov.

$$Ocena_{(atom/atom)} = 28155 + 4783 + 28924 + 307 \times 10 = 64932 \tag{6.1}$$

$$Ocena_{(Microsoft/vscode)} = 14881 + 1945 + 6178 + 109 \times 10 = 24094 \tag{6.2}$$

$$Ocena_{(facebook/nuclide)} = 4416 + 274 + 4753 + 79 \times 10 = 10233 \tag{6.3}$$

$$Ocena_{(adobe/brackets)} = 25423 + 5321 + 16890 + 295 \times 10 = 50584 \tag{6.4}$$

Vidimo lahko, da je po ocenah skupnosti na zadnjem mestu odprtokodni urejevalnik družbe Facebook, na predzadnjem mu sledi Microsoftov IDE. Drugo mesto pripada urejevalniku Brackets družbe Adobe. Na podlagi najvišje ocene skupnosti (6.1) ter prilagodljivost, dokumentacije izberemo urejevalnik Atom. Na svoj sistem lahko namestimo tudi ostale in jih uporabimo po želji. Po namestitvi urejevalnika bomo verjetno namestili še katerega izmed dodatnih paketov ter si urejevalnik tako dodatno prilagodili. Na voljo so nam tudi paketi, ki nam olajšajo pisanje datotek LaTeX.

6.1.4 Namestitev ogrodja Meteor

Meteor preprosto namestimo iz ukazne vrstice z ukazom:

```
1  curl https://install.meteor.com/ | sh
```

Na operacijskem sistemu Windows prenesemo in namestimo namestitveni program.

6.1.5 Uporaba OpenCV

Za namestitev OpenCV na računalniku Mac ni uradnih navodil v dokumentaciji OpenCV [55]. Najlažja možnost je tako vsaj navidez uporaba virtualnega računalnika.

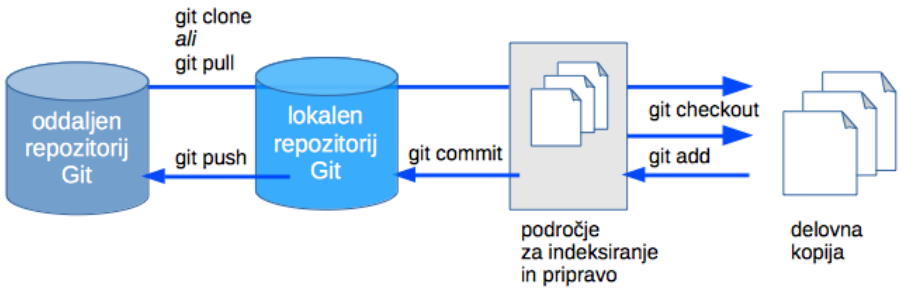
Za virtualizacijo izberemo odprtokodni *Virtual Box*, ki je na voljo za vse operacijske sisteme, tudi za Mac OS X. Pripravo operacijskega sistema na virtualnem računalniku si lahko olajšamo z orodji, ki omogočajo nameščanje iz shramb virtualnih naprav.

Prva možnost za pripravo virtualnega okoja so *virtualni stroji* (*angl. virtual machine, VM*). Ti se izvajajo na računalniku, običajno strežniku, kjer se na gostiteljskim operacijskim sistemu izvaja *hipervizor* (*angl. hypervisor*), ki skrbi za upravljanje virov in virtualnih strojev.

Predstavnik orodij za pripravo virtualnih strojev je *Vagrant*, ki deluje z virtualizacijo Virtual Box. Pripravo virtualnega stroja definiramo s pomočjo datoteke *Vagrantfile*, ki vsebuje vse ukaze, ki so potrebni za prenos in izvajanje ukazov ob zagonu virtualnega stroja z ukazom `vagrant up` .

Druga možnost je uporaba tehnologije *kontejnerjev* (*angl. containers*), kot je npr. tehnologija *Docker*. Ta omogoča pripravo virtualiziranega okolja v obliki kontejnerjev, ti so izolirani vendar souporabljajo operacijski sistem in kjer je to primerno, tudi binarno kodo in knjižnice. Postopek priprave določimo v datoteki *Dockerfile*, ki se izvede ob ukazu `docker build` .

Obe možnosti nudita tudi možnost uporabe repozitorijev. Pripravo virtualnega okolja lahko začnemo že z vnaprej pripravljenimi postopki. Osrednja



Slika 6.1: Prikaz upravljanja kode z ukazi Git [56].

knjižnica za povezovanje Node.js z OpenCV vsebuje tako datoteko Vagrantfile, kot datoteko Dockerfile.

6.1.6 Upravljanje izvorne kode

Med razvojem uporabimo distribuiran sistem za upravljanje z izvorno kodo Git, ta poskrbi za integriteto podatkov in podpira vzporedne, nelinearne tokove dela.

Če nimamo lastnega strežnika Git, najprej ustvarimo svoj račun na spletnem mestu GitHub, ter nov repozitorij, ki ga kloniramo v svoj razvojni računalnik z ukazom `git clone https://github.com/erikusaj/fri2016.git` .

Spremembe kode upravljamo s pomočjo ukazov [56] `git clone` in `git pull` za prevzem kode iz oddaljenega (*angl. remote*) strežnika Git v lokalno kopijo repozitorija (*angl. local*), ter ukazi `git add` , `git commit` in `git push` za objavo nazaj na oddaljen repozitorij (Slika 6.1).

Poglavje 7

Sklepne ugotovitve

Kompleksnost celostnega razvoja se veča

Celosten razvoj aplikacij se širi iz do običajnih dveh ključnih elementov spletne aplikacije, ki jo sestavljata odjemalski in strežniški del še na aplikacije za mobilne telefone in tablice. Tudi strežniško zaledje vse bolj prerašča in prehaja v zaledje v oblaku, tu so seveda še namizne aplikacije, ki so še vedno aktualne, ter servisi API, ki vnašajo dodatne plasti v verigi med odjemalcem in strežnikom. Na področje celostnega razvoja pa vse bolj vstopajo tudi povezane naprave IoT, ki še dodatno zakomplicirajo celotno problematiko.

Skupen ekosistem je ključen za obvladovanje kompleksnosti

Če na vsak element, gledamo kot na ločeno okolje, ki zahteva lastna razvojna orodja, okolje za razhroševanje in lasten programski jezik postane vse skupaj neverjetno kompleksno in za razvijalce težko obvladljivo. Kar nam lahko pomaga do rešitve kompleksne situacije je bolj poenotena platforma, ki se čim bolj obnese v vseh teh posamičnih okoljih in s tem doprinese enoten ekosistem v vsa okolja. Na Node.js lahko gledamo kot poenoteno platformo in skupen ekosistem.

Danes praktično ni predstavnika industrije IT, ki nima vsaj manjšega projekta temelječega na JS. Tudi podjetja, ki so še pred kratkim v javnih izjavah dajale prednost razvoju domorodnih mobilnih aplikacij (*angl. native*

mobile applications), tu imamo v mislih predvsem Facebook, ki danes precejšnji del razvoja usmerjajo tudi v razvoj tehnologij in ekosistemov, katerih ključni element je JS.

Izjema je morda le družba Apple, kar je iz vidika ustvarjanja dodane vrednosti lastnih mobilnih naprav in ekosistema okrog le-teh povsem razumljivo, po drugi strani pa se izkaže, da so prav računalniki Apple Mac pri razvijalcih zelo cenjeni kot ključno orodje za celosten odjemalsko-strežniški razvoj aplikacij, če lahko sklepamo po računalnikih, ki jih predavatelji uporabljajo za predstavitve in praktične prikaze na različnih konferencah.

Pomen skupnost razvijalcev

Za razvijalca-posameznika je razvoj celostnega odjemalsko-strežniškega sklada težko dosegljiv cilj. Poenotenje skladov tehnologih v smislu enotnega programskega jezika lahko sicer poenostavi in zmanjša napor in zahtevana znanja, a navkljub temu nihče ne pričakuje, da bo lahko posameznik res kakovostno kos kompleksnejšemu problemu. Posamezniku je lahko v pomoč skupnost razvijalcev, ki razvijajo ali so razvili posamezne module, ki jih lahko ostali razvijalci znova uporabimo pri reševanju drugega problema. Strateški izbor sklada tehnologij in ogrodij in nenazadnje sposobnost analize in izbire uporabljenih modulov lahko posamezniku ali podjetju pomaga, da občutno skrajša čas, ki je potreben za implementacijo rešitve zahtevnejšega problema.

Celostni razvoj in sodobna podjetja

Razvijalci celostnih odjemalsko-strežniških rešitev, so tako ključni za podjetja, predvsem iz vidika interne mobilnosti delovnih mest. Z interno mobilnostjo lahko podjetje tekom projektov določeno število razvijalcev ustrezno prerazporedi na posamezne dele razvoja, kjer so človek-ure predstavljajo najbolj kritičen izziv. Ključno vlogo pri tem ima skupni jezik, saj tovrstne prerazporeditve ne pridejo več v poštev, če je pred tem zahtevano nekaj mesečno usposabljanje za novo tehnologijo na drugi poziciji. Sodobno podjetje, ki se ukvarja z razvojem programske opreme bodisi strežniških, splet-

nih ali mobilnih aplikacij mora najprej analizirati minimalne skupne imenovalce in postopke za katere so usposobljeni vsi razvijalci. Običajno so skupni imenovalci vezani na same procese, od specifikacij do upravljanja in vodenja razvoja, testiranja, upravljanja napak in podobno. Če je poleg naštetega eden izmed izbranih skupnih imenovalcev tudi skupen primeren programski jezik je to lahko še dodatna prednost za agilnost razvojnih skupine v podjetju.

Odprtost in svoboden pretok znanja

Tudi omejevanje na eno samo tehnologijo ni dobro. Podjetje, ki se ukvarja s sodobnimi tehnologijami razvoja aplikacij bi moralo zato predvsem skrbeti za klimo, ki omogoča svoboden pretok informacij in znanja, predvsem odprto in strokovno analizo argumentov za in proti. Če čas to dopušča, tudi možnost primerjave in vzpostavitve dveh ali več prototipov na različnih skladih tehnologij, kjer se na podlagi dejanskih rezultatov lahko izbere v danem trenutku boljša tehnologija, ki omogoča ustrezne preformančne rezultate in je tudi is stroškovnega vidika zahtevanih razvojnih človek-ur in ocene vzdrževanja primernejša.

Razvoj lastnih storitev za računalniški vid

Danes imamo na voljo številne storitve v oblaku za računalniški vid in umetno inteligenco (*artificial inteligence, AI*): Microsoft Cognitive Services [57], IBM Visual Recognition [62], Google Cloud Vision API [63], itd. Vse omogočajo enostavno, hitro in cenovno ugodno implementacijo računalniškega vida in možnost izdelave klasifikatorjev slik.

Spoznali smo tudi, da je implementacija odprtokodnih knjižnic enostavna in dostopna tudi v JS. Na voljo imamo tudi knjižnice, ki omogočajo razvoj AI: sta npr. Mind [64] in vmesnik JS za TensorFlow [65], Googlovo odprtokodno knjižnico AI [66]. Zavedati se moramo, da so inovacije možne le z znanjem in lastnim razvojem, ki je s pomočjo odprtokodnih komponent dostopnejši. Komercialne rešitve so nam lahko v oporo in obenem predstavljajo odličen primerjali preskus za lastne storitve.

Literatura

[1] J. J. "Garret Ajax: A New Approach to Web Applications" [Online]. Dosegljivo: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/. [Dostopano 30. 5. 2016].

[2] D. Flanagan. "JavaScript: The Definitive Guide", str 1. 2011.

[3] "Uvod v JSON" [Online]. Dosegljivo: http://json.org/json-sl.html [Dostopano 30. 5. 2016].

[4] ECMA. "ECMA-404 - The JSON Data Interchange Format" [Online]. Dosegljivo: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf [Dostopano 30. 5. 2016].

[5] IETF. "The application/json Media Type for JavaScript Object Notation (JSON)" [Online]. Dosegljivo: http://www.ietf.org/rfc/rfc4627.txt [Dostopano 30. 5. 2016].

[6] E. McKean. "Fluent 2015: The Linguistics of JavaScript" [Online][Video file]. Dosegljivo: https://youtu.be/4sNUzqrhQqY?t=1m30s [Dostopano 30. 5. 2016].

[7] W3C. "A Short History of JavaScript" [Online]. Dosegljivo: https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript. [Dostopano 30. 5. 2016].

[8] Ecma International. "ECMAScript: A general purpose, cross-platform programming language" [Online]. Dosegljivo: http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf. str. 1-2, 1997 [Dostopano 30. 5. 2016].

[9] Ecma International. "Standard ECMA-262 ECMAScript® 2015 Language Specification" [Online]. Dosegljivo: http://www.ecma-international.org/publications/standards/Ecma-262-arch.htm. [Dostopano 30. 5. 2016].

[10] kangax, webbedspace, zloirock. "ECMAScript compatibility table" [Online]. Dosegljivo: http://kangax.github.io/compat-table/es5/ [Dostopano 30. 5. 2016].

[11] Ecma International. "Standard ECMA-262 5.1 Edition" 15.9.5.44. [Online]. Dosegljivo: http://www.ecma-international.org/ecma-262/5.1/#sec-15.12 [Dostopano 30. 5. 2016].

[12] Ecma International. "ECMAScript: A general purpose, cross-platform programming language" [Online]. Dosegljivo: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. [Dostopano 30. 5. 2016].

[13] Mozilla. "Classes - JavaScript" [Online]. Dosegljivo: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes. [Dostopano 30. 5. 2016].

[14] Microsoft. "TypeScript - JavaScript that scales" [Online]. Dosegljivo: https://www.typescriptlang.org/ [Dostopano 30. 5. 2016].

[15] T. Schlossnagle “Node-amqp is an AMQP client for nodejs”. [Online]. Dosegljivo:
`https://github.com/postwait/node-amqp`. [Dostopano 30. 5. 2016].

[16] OASIS “Advanced Message Queuing Protocol”. [Online]. Dosegljivo:
`https://www.amqp.org/`. [Dostopano 30. 5. 2016].

[17] Linux Foundation Collaborative Project. “Open API Initiative”. [Online]. Dosegljivo:
`https://openapis.org/` [Dostopano 30. 5. 2016].

[18] StrongLoop, IBM. “LoopBack - The Node.js API Framework”. [Online]. Dosegljivo:
`http://loopback.io/` [Dostopano 30. 5. 2016].

[19] S. Khan, Prof. V. Mane. International Journal of Scientific and Research Publications, Volume 3, Issue 10, October 2013. “SQL Support over MongoDB using Metadata”. [Online]. Dosegljivo:
`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.395.8668&rep=rep1&type=pdf` [Dostopano 30. 5. 2016].

[20] K. Chodorow, M. Dirolf. “MongoDB, The Definitive Guide”

[21] K. E. Wiegers “Software Requirements”, str 234. 2003.

[22] GitHub.com “GitHub” [Online]. Dosegljivo:
`https://www.github.com`. [Dostopano 30. 5. 2016].

[23] Microsoft Openness Team. “Bash on Ubuntu on Windows” [Online]. Dosegljivo:
`http://openness.microsoft.com/blog/2016/04/04/bash-ubuntu-windows/`. [Dostopano 30. 5. 2016].

[24] Wikipedia. “ISO 8601” [Online]. Dosegljivo:
`https://sl.wikipedia.org/wiki/ISO_8601` [Dostopano 30. 5. 2016].

[25] S. Hanselman. "On the nightmare that is JSON Dates." [Online]. Dosegljivo: http://www.hanselman.com/blog/ [Dostopano 30. 5. 2016].

[26] M. Pilgrim. "Dive Into Python 3", poglavje 13. podpoglavje 10. "Serializing Datatypes Unsupported by json" [Online]. Dosegljivo: http://www.diveintopython3.net/serializing.html#json-unknown-types [Dostopano 30. 5. 2016].

[27] MongoDB. "BSON - Binary JSON" [Online]. Dosegljivo: http://bsonspec.org/. [Dostopano 30. 5. 2016].

[28] MongoDB. "Aggregation Pipeline Operators" [Online]. Dosegljivo: https://docs.mongodb.com/manual/reference/operator/aggregation/#aggregation-pipeline-operator-reference [Dostopano 30. 5. 2016].

[29] Node.js Foundation, Strongloop, IBM. "Express" [Online]. Dosegljivo: http://expressjs.com/ [Dostopano 30. 5. 2016].

[30] Google. "AngularJS: Developer Guide" [Online]. Dosegljivo: https://docs.angularjs.org/guide/introduction [Dostopano 30. 5. 2016].

[31] V. Patel. "AngularJS: A Modern MVC Framework in JavaScriptAngularJS Service / Factory Tutorial with Example" [Online]. Dosegljivo: http://viralpatel.net/blogs/angularjs-service-factory-tutorial/ [Dostopano 30. 5. 2016].

[32] N. Jain. P. Mangal. D. Mehta. "AngularJS: A Modern MVC Framework in JavaScript" [Online]. Dosegljivo: http://www.jgrcs.info/index.php/jgrcs/article/viewFile/952/610 [Dostopano 30. 5. 2016].

[33] Node.js Foundation. "Node.js" [Online]. Dosegljivo: https://nodejs.org/en/. [Dostopano 30. 5. 2016].

[34] Modulecounts. "Modulecounts" [Online]. Dosegljivo: http://www.modulecounts.com/. [Dostopano 30. 5. 2016].

[35] IDC. "Smartphone OS Market Share, 2015 Q2" [Online]. Dosegljivo: http://www.idc.com/prodserv/smartphone-os-market-share.jsp. [Dostopano 30. 5. 2016].

[36] W3C. "Mobile Web Application Best Practices" [Online]. Dosegljivo: https://www.w3.org/TR/mwabp/#bp-presentation-perceived [Dostopano 30. 5. 2016].

[37] J. Nielsen. "Usability 101: Introduction to Usability" https://www.nngroup.com/articles/usability-101-introduction-to-usability/ [Dostopano 30. 5. 2016].

[38] E. Lamprecht (2016) "The Difference Between UX and UI Design-A Layman's Guide" Dosegljivo: http://blog.careerfoundry.com/ui-design/the-difference-between-ux-and-ui-design-a-laymans-guide/ [Dostopano 23. 5. 2016].

[39] "Callback Hell" [Online]. Dosegljivo: http://callbackhell.com/. [Dostopano 30. 5. 2016].

[40] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures", poglavje 5. "Representational State Transfer (REST)" [Online]. Dosegljivo: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm [Dostopano 30. 5. 2016].

[41] Wikipedia. "Representational state transfer" [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Representational_state_transfer [Dostopano 30. 5. 2016].

[42] A Facebook & Instagram collaboration. “Why React” [Online]. Dosegljivo: https://facebook.github.io/react/docs/why-react.html. [Dostopano 30. 5. 2016].

[43] Apache. “Documentation - Apache Cordova” [Online]. Dosegljivo: https://cordova.apache.org/docs/en/latest/ [Dostopano 30. 5. 2016].

[44] Apache Cordova. “Cordova Plugin Search” [Online]. Dosegljivo: https://cordova.apache.org/plugins/ [Dostopano 30. 5. 2016].

[45] W3C, “Geo API” [Online]. Dosegljivo: http://dev.w3.org/geo/api/spec-source.html. [Dostopano 30. 5. 2016].

[46] W3C, “Vibration API” [Online]. Dosegljivo: https://www.w3.org/TR/vibration/. [Dostopano 30. 5. 2016].

[47] Kadira. “Mantra” [Online]. Dosegljivo: https://kadirahq.github.io/mantra/. [Dostopano 30. 5. 2016].

[48] Wikipedia, “Chromium Embedded Framework - Applications using CEF” [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Chromium_Embedded_Framework#Applications_using_CEF [Dostopano 30. 5. 2016].

[49] Node.js Foundation “New Node.js Foundation Survey Reports New “Full Stack” In Demand Among Enterprise Developers” [Online]. Dosegljivo: https://nodejs.org/en/blog/announcements/nodejs-foundation-survey/. [Dostopano 30. 5. 2016].

[50] The Linux Foundation Collaborative Projects “Node.js 2016 User Survey Report” [Online]. Dosegljivo:

https://nodejs.org/static/documents/2016-survey-report.pdf. [Dostopano 30. 5. 2016].

[51] IBM Emerging Technologies, "Node-RED" [Online]. Dosegljivo: http://nodered.org//. [Dostopano 30. 5. 2016].

[52] Apple. "Apple - Open Source" [Online]. Dosegljivo: http://www.apple.com/opensource/. [Dostopano 30. 5. 2016].

[53] Apple. "How to install Windows using Boot Camp" [Online]. Dosegljivo: https://support.apple.com/en-us/HT201468 [Dostopano 30. 5. 2016].

[54] P. Braden. "OpenCV Bindings for node.js" [Online]. Dosegljivo: https://github.com/peterbraden/node-opencv/blob/master/README.md [Dostopano 30. 5. 2016].

[55] OpenCV. "Introduction to OpenCV" [Online]. Dosegljivo: http://docs.opencv.org/2.4/doc/tutorials/introduction/table_of_content_introduction/table_of_content_introduction.html [Dostopano 30. 5. 2016].

[56] Git. "Git - Reference" [Online]. Dosegljivo: https://git-scm.com/docs [Dostopano 30. 5. 2016].

[57] Microsoft. "Cognitive Services - Computer Vision API" [Online]. Dosegljivo: https://www.microsoft.com/cognitive-services/en-us/computer-vision-api [Dostopano 30. 5. 2016].

[58] A. M. Øygard. "clmtrackr" [Online]. Dosegljivo: https://github.com/auduno/clmtrackr/blob/dev/README.md [Dostopano 30. 5. 2016].

[59] E. Zatepyakin. "Jsfeat - JavaScript Computer Vision library" [Online]. Dosegljivo:

https://inspirit.github.io/jsfeat/[Dostopano 30. 5. 2016]. [Dostopano 30. 5. 2016].

[60] O. Smith. "face-detect" [Online]. Dosegljivo: https://github.com/orls/ccv-purejs/blob/master/README.md [Dostopano 30. 5. 2016].

[61] E. Lundgren, T. Rocha Z. Rocha P. Carvalho M. Bello. "tracking.js - A modern approach for Computer Vision on the web" [Online]. Dosegljivo: https://trackingjs.com/ [Dostopano 30. 5. 2016].

[62] IBM. "Visual Recognition - Understand the contents of images. Create custom classifiers to develop smart applications." [Online]. Dosegljivo: http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/visual-recognition.html [Dostopano 30. 5. 2016].

[63] Google. "Cloud Vision API - Derive insight from images with our powerful Cloud Vision API" [Online]. Dosegljivo: https://cloud.google.com/vision/ [Dostopano 30. 5. 2016].

[64] S. Miller. "Mind - Flexible neural networks in JavaScript" [Online]. Dosegljivo: https://github.com/stevenmiller888/mind [Dostopano 30. 5. 2016].

[65] N. Kothari. "TensorFlow + Node.js" [Online]. Dosegljivo: https://github.com/nikhilk/node-tensorflow [Dostopano 30. 5. 2016].

[66] Google. "TensorFlow - an Open Source Software Library for Machine Intelligence" [Online]. Dosegljivo: https://www.tensorflow.org/ [Dostopano 30. 5. 2016].

[67] Mozilla Developer Network. "JavaScript" [Online]. Dosegljivo: https://developer.mozilla.org/en-US/docs/Web/JavaScript [Dostopano 30. 5. 2016].

[68] GNU General Public Licence. [Online]. Dosegljivo: `https://www.gnu.org/copyleft/gpl.html`. [Dostopano 30. 5. 2016].