Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Zidarn

Razvoj spletne aplikacije za urejanje slik

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Aleš Smrdel

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Pogosto imamo na voljo slike na računalnikih ali pa mobilnih napravah, ki bi jih radi varno hranili in urejali pa nimamo na voljo ustreznega orodja. V okviru diplome razvijte spletno aplikacijo, ki bo registriranemu uporabniku omogočala shranjevanje slik na strežnik ter urejanje slik, shranjenih na strežniku. Polge tega omogočite tudi deljenje slik na strežniku z drugimi uporabniki. Razvijte tudi mobilno aplikacijo, ki bo omogočala prijavo na strežnik, fotografiranje in neposredno shranjevanje posnetih fotografij oziroma slik na strežnik. Pri razvoju aplikacije izberite ustrezne tehnologije na strani strežnika in odjemalca, prav tako pa izberite tudi primerno mobilno platformo. Poskrbite tudi za objavo aplikacije na javnem strežniku.

Rad bi se zahvalil staršema, ki sta me vedno podpirala in, ko so sem bil na tleh sta mi vedno pomagala, da sem se pobral. Zahvalil bi se tudi mojemu mentorju doc. dr. Alešu Smrdelu za potrpežljivost in pomoč pri izdelavi diplomske naloge.

What hurts more, the pain of hard work or the pain of regret?

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ROR	Ruby on Rails	Ruby on Rails
HTML	Hyper Text Markup Language	Jezik za označevanje nadbesedila
CSS	Cascading Style Sheets	Kaskadne stilske predloge
AJAX	Asynchronous JavaScript and XML	Asinhroni JavaScript in XML
XML	Extensible Markup Language	Razširljiv označevalni jezik
DOM	Document Object Model	Objektni model dokumenta
HTTP	Hyper Text Transfer Protocol	Protokol za izmenjavo nadbesedila
JSON	JavaScript Object Notation	JavaScript objekt za prenašanje podatkov
REST	Representational State Transfer	Prenos predstavitvenega stanja
MVC	Model-View-Controller	Model-Pogled-Nadzornik
SQL	Structured Query Language	Strukturirani povpraševalni jezik
URL	Uniform Resource Locator	Enolični krajevnik vira
DRY	don't repeat yourself	ne ponavljaj se
ORM	Object-Relational Mapping	Objektno-relacijska preslikava
API	Application Programming Interface	Aplikacijski programski vmesnik
PNG	Portable Network Graphics	slikovni format
JPG	Joint Photographic Group	slikovni format
GIF	Graphics Interchange Format	slikovni format
PDF	Portable Document Format	dokumentni format

Povzetek

Naslov: Razvoj spletne aplikacije za urejanje slik

Avtor: Rok Zidarn

Danes sta pojma splet in aplikacije tesno povezana. Njuno stičišče so spletne aplikacije, ki uporabniku omogočajo izvajanje različnih aktivnosti, oziroma na splošno opravljanje dela, kar preko spleta. Na nivoju aplikacijskega sloja komunikacija pogosto poteka preko protokola HTTP in njegovih osnovnih metod, kot so GET, POST, PUT, DELETE ter HEAD. Ker pa se danes promet na spletu vse bolj seli na mobilne naprave, kot so tablice in pametni telefoni, je priporočljivo uporabniku omogočiti tudi dostop preko teh naprav. Namen diplome oziroma cilj je razvoj spletne aplikacije, ki omogoča prijavo uporabnika, nalaganje ter urejanje slik oziroma fotografij. Pri tem pa bo urejanje potekalo preko zahtevka strežniku, ta pa bo klical ustrezno funkcijo v orodju ImageMagick. Uporabniku pa bo na voljo tudi mobilna aplikacija, ki bo omogočala neposredno shranjevanje posnetih fotografij z mobilnega telefona ali tablice na njegov račun.

Ključne besede: spletna aplikacija, urejanje, slika, Ruby on Rails, ImageMagick.

Abstract

Title: Development of a web application for image editing

Author: Rok Zidarn

Today, both the concept of the web and of applications are heavily intertwined. Their union is embodied by web applications that allow the user to perform various operations, or more generally, do work over the web. On the application layer, communication often uses the HTTP protocol via its basic methods (GET, POST, PUT, DELETE and HEAD). Since web traffic is slowly moving from desktop computers to smart phones and tablets, it is advisable to build web applications with these devices in mind. The main purpose or goal of this thesis is the development of a web application that will allow users to register, upload and edit pictures or photos. The editing will be triggered with a request to the server, which will call a specific function in a program called ImageMagick. A mobile app will be also available to the users, which will enable direct storing of photos, taken with their mobile phone or tablet, in their account.

Keywords: web application, editing, image, Ruby on Rails, ImageMagick.

Poglavje 1

Uvod

Diplomska naloga predstavlja razvoj spletne aplikacije ter mobilne aplikacije, ki omogoča nalaganje slik na strežnik, urejanje slik na strežniku ali na lokalnem računalniku ter hranjenje slik. Tematika je aktualna predvsem zaradi tega, ker živimo v času socialnih omrežij in deljenja multimedijskih vsebin, kot so na primer fotografije in tudi video posnetki. Ravno zaradi tega vsak, ki posname fotografijo in jo ima namen objaviti, želi, da bi fotografija izgledala čim bolj profesionalno, brez napak. Tukaj pride prav aplikacija, ki bo omogočala hitre popravke, in ne bo zahtevala pretiranega dela ter znanja. Poleg tega pa bi lahko postala tudi alternativa drugim precej bolj kompleksnim in plačljivim aplikacijam kot je na primer Adobe Photoshop [1].

Pri razvoju bomo uporabili ogrodje Ruby on Rails [24], ki bo zagotavljalo komunikacijo med odjemalcem, strežnikom ter relacijsko podatkovno bazo. Za hranjenje slik bomo uporabili oblačno storitev Amazon Web Service (AWS) Simple Storage Service (S3) [3], urejanje pa bo omogočalo orodje ImageMagick [25]. Uporabniku bo treba ponuditi čim več različnih in uporabnih slikovnih učinkov nad slikami, od filtrov preko upravljanja barv, do raznih drugih orodij za urejanje slik. Da bo olajšano shranjevanje z mobilnih naprav, kot so telefoni in tablice, bo na voljo tudi mobilna aplikacija, s pomočjo katere se bodo lahko neposredno nalagale fotografije z naprave na uporabnikov račun.

Aplikacija je razvita v naslednjih tehnologijah: Hyper Text Markup Language 5 (HTML5), JavaScript (JS), Cascading Style Sheets (CSS), Asynchronous JavaScript And XML (AJAX), Ruby, Ruby on Rails (ROR), ImageMagick in Android OS, oziroma programski jezik Java, ki se uporablja za razvoj aplikacij na platformi Android. Osnova celotne aplikacije je spisana v programskem jeziku Ruby, ki se predvsem uporablja pri razvoju spletnih aplikacij. Pri samem razvoju aplikacije pa smo uporabili ogrodje Ruby on Rails, kar precej olajša delo in razvoj. Za prikaze vsebine in za samo strukturo spletne aplikacije smo uporabili zadnjo, peto verzijo standarda označevalnega jezika HTML. Sama oblika (angl. design) aplikacije je dodelana s pomočjo CSS-ja, za dinamičnost pa skrbi JavaScript. Pri komunikaciji s strežnikom smo uporabili skupek tehnologij AJAX, ki omogoča asinhrone zahtevke, uporabljene predvsem pri klicu funkcij orodja ImageMagick. Asinhroni zahtevki omogočajo ponovno nalaganje le dela HTML dokumenta, zaradi česar je delovanje aplikacije hitrejše, uporabniška izkušnja pa boljša. Za poizvedbe nad bazo se uporablja Structured Query Language (SQL), vendar glede na to, da ogrodje ROR podpira objektno-relacijsko mapiranje (ORM), ni bilo potrebno pisanje dodatnih SQL poizved, saj za njih poskrbi že ORM komponenta ogrodja ROR. Za urejanje slik smo uporabili orodje ImageMagick, ki omogoča urejanje slik, ki je dostopno preko Ruby gem-a mini_magick.

Za strežniško ogrodje Ruby on Rails [23] sem se odločil predvsem zaradi dveh razlogov. Prvi je, da sem v sklopu svojega študija v okviru dveh projektnih nalog že sprogramiral dve spletni aplikaciji v jeziku PHP. Zaradi tega sem si še zaželel, da bi se še naučil kakšne dodatne tehnologije za razvoj spletnih aplikacij. Ko sem pregledoval, katere spletne tehnologije se najbolj uporabljajo, sem ugotovil, da je na prvem mestu PHP, sledi pa mu ASP.NET, nad katerim pa nisem bil najbolj navdušen [22]. Med samim odločanjem sem spoznal tudi ogrodje Spring, ki se uporablja pri javanskih spletnih tehnologijah. Ker sem si še vedno želel spoznati kakšno novo tehnologijo, sem se odločil za Ruby. Poleg tega pa je bil ravno Ruby on Rails tretji na seznamu najbolj priljubljenih spletnih tehnologij. Kot drugi razlog

pa bi navedel predavanje Jana Berdajsa, ki me je prepričal v izbiro Ruby-ja. Med raziskovanjem različnih tehnologij pa sem tudi izvedel, da je bil Twitter razvit v tej tehnologiji, kar je le potrdilo mojo končno odločitev.

Preden smo se dejansko lotili razvoja aplikacije, smo malce pregledali področje oziroma konkurenco, torej kaj ponujajo drugi. Ugotovili smo, da večina podobnih aplikacij, ki smo jih pregledali, bodisi ponujajo zgolj urejanje in ne trajne hrambe bodisi le trajno hrambo brez možnosti urejanja. Pod prvo skupino spadata aplikaciji kot sta Flickr [8] in Imgur [13], ki ponujata le hrambo slik, vendar imata za razliko od naše aplikacije tudi možnost komentiranja, všečkanja in označevanja (tag-anja) slik. Imgur ponuja sicer tudi osnovne možnosti urejanja slik, vendar le majhen nabor zmožnosti, bolj ali manj le nekaj osnovnih slikovnih učinkov. Pod drugo skupino aplikacij pa spadata Ipiccy [15] in Pixlr [19], ki ponujata predvsem urejanje. Ipiccy ima podobne možnosti urejanja kot naša aplikacija, vendar ne omogoča trajne hrambe, omogoča pa objavo na družabnem omrežju Facebook. Pixlr pa podpira precej bolj kompleksne slikovne učinke in možnosti urejanja, ki jih naša aplikacija ne omogoča, kot so zgodovina (angl. undo, redo), risanje in sloji (angl. layers).

V naslednjem poglavju bomo natančneje opisali tehnologije, s pomočjo katerih je razvita spletna aplikacija. Najprej bomo opisovali tehnologije na strani odjemalca kot so HTML, CSS in AJAX. Nato bomo opisali še tehnologije na strani strežnika, kjer bomo opisali principe programskega jezika Ruby, ogrodje Ruby on Rails ter sintakso in delovanje orodja ImageMagick. Poleg tega bomo predstavili tudi osnove razvoja mobilnih aplikacij za operacijski sistem Android. V tretjem poglavju bomo natančneje opisali razvoj aplikacije v teh tehnologijah. V četrtem poglavju bomo predstavili delovanje spletne in mobilne aplikacije. Za konec pa bomo predstavili še zaključke in opis nadaljnega dela.

Poglavje 2

Uporabljene tehnologije in orodja

V tem poglavju bomo predstavili tehnologije, ki smo jih uporabili pri razvoju spletne in mobilne aplikacije. Na začetku lahko poudarimo, da se, glede na lokacijo uporabe, spletne tehnologije delijo na dve vrsti. Tiste, ki se uporabljajo na strani odjemalca, kot so HTML, CSS, JavaScript in podobno, ter na tiste, ki jim pravimo strežniške (angl. server-side) tehnologije oziroma drugače rečeno tehnologije na strani strežnika. V našem primeru sta to programski jezik Ruby in ogrodje Ruby on Rails. Urejanje slik v naši aplikaciji pa omogoča orodje ImageMagick. Uporaba tega orodja poteka s pomočjo Ruby gem-a mini_magick, ki se ga kliče na strani strežnika.

Ena izmed nalog, ki smo si jih zadali je tudi objava spletne aplikacije na spletu. Za to smo uporabili platformo Heroku, ki je olajšala objavo naše aplikacije, zraven pa smo uporabili tudi oblačno storitev AWS S3, ki podpira trajno hranjenje slik, ki jih nalagajo uporabniki na svoj račun.

Poleg spletne aplikacije smo razvili tudi podporno mobilno aplikacijo v programskem jeziku Java, namenjeno zgolj Android napravam. Za verzioniranje programske kode pa smo poskrbeli z orodjem GitBash ter repozitorijem na platformi GitHub.

2.1 Tehnologije na strani odjemalca

Najprej bomo predstavili tehnologije, ki skrbijo za prikaz vsebine na strani odjemalca.

2.1.1 HTML

Hyper Text Markup Language (HTML) je tehnologija, ki se uporablja na strani odjemalca. Namenjena je prikazu in strukturiranju podatkov na spletni strani oziroma aplikaciji. Definiran je s pomočjo Standard Generalized Markup Language-a (SGML), ki predstavlja standard tvorjenja označevalnih jezikov.

Osnovni gradnik s katerim je definirana tehnologija HTML je značka, ki je objeta z dvema znakoma in črko ali besedo znotraj njih. Nek element je sestavljen iz dveh značk, začetne in zaključne značke, ki definirata način prikaza, ter vsebine, ki je zapisana med njima in je namenjena prikazu. Poznamo tudi nekaj posebnih elementov, ki ne morejo imeti vsebine. Tovrstni elementi so sestavljeni samo iz ene značke, pri tem imajo začetno in končno značko združeno, kot na primer značka za dodajanje slik (*img* HTML element v primeru 2.1). Poleg značk in vsebine lahko HTML dokumentu določimo tudi atribut, kot je na primer vir slike (atribut *src*), ki jo želimo prikazati, ali pa mu določimo stil (atribut *style*).

Primer 2.1: Osnovni HTML dokument

```
<html>
   <head>
      <title>Spletna stran</title>
   </head>
   <body>
      <h1>NASLOV</h1>
      <p>Besedilo odstavka</p>
      <a href="www.google.com">Povezava</a>
      <img src="/doc/image1.png" alt="slika"/>
```

```
    <p style="color: blue;">Moder odstavek</p>
  </body>
</html>
```

Programi, ki so namenjeni prikazu vsebine, ki je definirana v HTML dokumentu, so brskalniki, kot so na primer Google Chrome, Mozilla Firefox, Microsoft Edge in Opera.

Trenutno najnovejši standard je HTML5, ki je prav tako uporabljen v naši aplikaciji. Podpira pa določene novosti, kot so: dodatni/novi tipi vnosnih polj (barva, datum, čas, podpora multimedijskih vsebin, orodje za grafiko - canvas), podpora mobilnim napravam in tako naprej.

2.1.2 CSS

Tehnologija Cascading Style Sheets (CSS) je tehnologija namenjena oblikovanju grafične podobe vsebine, ki je zapisana znotraj HTML elementov. To je podporna tehnologija, nastala pa je z namenom ločevanja dela programerjev (struktura dokumenta HTML) in dela oblikovalcev (videz dokumenta HTML).

CSS pravila lahko zapisujemo v posebej definiranih datotekah, lahko pa v samem HTML dokumentu v njegovi glavi, ali pa kar znotraj HTML elementov v atributu *style*, obstaja pa tudi možnost spreminjanja preko programske kode v JavaScript-u.

CSS nam omogoča oblikovanje stila v sedmih kategorijah: Pisava, seznami, barve, poravnana, robovi, obrobe in ozadje. Stil lahko določimo množici elementov, ki ga definiramo na različne načine. Prvi način omogoča določanje stila kar vsem enakim HTML elementom. V primeru 2.2 imajo vsi elementi h1 pisavo velikosti dvanajst slikovnih elementov (angl. pixel). Lahko določimo stil točno določenemu elementu na podlagi identifikacijskega niza (id, *#slika* v primeru 2.2). V tem primeru smo določili 50% širino elementu katerega atribut *id* ima vrednost slika. Pri tem načinu izbire množice elementov je treba poudariti, da lahko s tem stil določimo natanko enemu

elementu. Če pa želimo določiti stil večim različnim HTML elementom pa uporabimo razred (*.avtomobili* v primeru 2.2). V tem primeru pa imajo vsi elementi, ki sodijo v razred avtomobili, rdečo polno obrobo širine dveh slikovnih elementov.

Primer 2.2: CSS dokument

```
h1 { font-size: 12px; }
#slika { width: 50%; }
.avtomobili { border: 2px solid red; }
```

2.1.3 JavaScript

JavaScript je programski jezik, ki se interpretira na strani odjemalca oziroma klienta. Uporablja se predvsem za izboljšanje uporabniške izkušnje, kar se tiče hitrosti, dinamičnosti in izgleda spletne strani oziroma aplikacije. S pomočjo JavaScript-a lahko preko Document-Object Model-a (DOM) predstavitve dokumenta dostopamo do HTML elementov in jih spreminjamo. Kar se tiče same sintakse je podoben programskemu jeziku Java, vendar se tu podobnost med jezikoma konča.

Od Jave se razlikuje po tem, da je dinamično tipiziran, kar pomeni, da se tip spremenljivke določi med izvajanjem. Poleg tega ne pozna razredov, omogoča pa prototipno dedovanje. Njegovo kodo lahko hranimo v posebni datoteki, lahko pa jo kar vključimo v HTML dokument znotraj *script* elementa. Zelo prav pa tudi pride pri skupku tehnologij AJAX, ki predstavlja povezavo XML tehnologije, JavaScript programske kode ter asinhronih klicev na strežnik.

S HTML dokumentom komunicira s pomočjo drevesne predstavitve dokumenta DOM, ki predstavlja programski vmesnik med HTML dokumentom in JavaScript kodo. Posledično lahko na primer spremenimo barvo oznak, dodamo nek razred na HTML element z določenim id-jem ali morda celo izpišemo neko opozorilo pri validaciji. Primer upravljanja s HTML elementi je prikazano v primeru 2.3 in poteka tako, da najprej pridobimo nek element

na podlagi njegovega id-ja. Nato pa elementu lahko dodamo poslušalca za različne dogodke nad tem elementom, kot so na primer premik miškinega kazalca, pritisk, držanje neke tipke na tipkovnici, izguba fokusa in tako naprej. Parameter, ki je za tem, predstavlja ime rokovalnika dogodka, ki se začne izvajati ob konkretnem dogodku.

Primer 2.3: Primer JavaScript programske kode

```
document.getElementById("gumb");
gumb.addEventListener("click", funkcijaPrikaz);
function funkcijaPrikaz() {
   document.getElementById("naslov").innerHTML = "POZDRAVLJENI";
}
```

jQuery [17] je ena izmed knjižnic, napisanih v JavaScript-u in je namenjena lažjemu ter hitrejšemu delu. Osnovna sintaksa pri uporabi knjižnice ima obliko *$selektor.akcija(){ // koda*}, ki hkrati predstavlja kateremu elementu (*$selektor*) in ob katerem dogodku (*akcija()*) se izvede želena koda (*//koda*). Prav tako pa omogoča tudi kompleksnejše zadeve, kot so animacije in dinamični meniji.

2.1.4 XML

Extensible Markup Language (XML) je prav tako označevalni jezik razvit z uporabo SGML-ja. Namenjen pa je predvsem zapisu podatkov ter njihovi hierarhiji in ne samemu prikazu, za razliko od HTML-ja. XML je namenjen temu, da si sam uporabnik definira nabor značk, s katerimi označi pomen vsebine, ki je shranjena v teh značkah. V osnovi je XML poenostavljena oblika SGML-ja, ki je namenjen pisanju označevalnih jezikov.

Najpogosteje ga zapišemo bodisi s pomočjo DTD-ja (Document Type Definition) ali XML sheme. Pri razčlenjevanju dokumenta XML poznamo dve vrsti pravilnosti dokumenta, in sicer dobro tvorjen (angl. well-formed) dokument, kar pomeni, da je dokument v skladu s pravili tvorjenja XML-ja in veljaven (angl. valid), kar pomeni, da dokument ustreza neki aplikaciji XML

oziroma naboru značk, ki jih določen jezik XML definira. V našem primeru se uporablja pri AJAX-u in pri oblikovanju grafičnega vmesnika Android mobilne aplikacije.

2.1.5 AJAX

AJAX je zbirka tehnologij na strani odjemalca, ki združuje XML in JavaScript. Sama zbirka tehnologij pa omogoča asinhrono komunikacijo s strežnikom. Asinhrona komunikacija pa omogoča boljšo uporabnikovo izkušnjo in hitrejše delovanje. Glavni namen je posodabljanje le dela HTML dokumenta, namesto ponovnega nalaganja celotne strani.

Osnovno delovanje poteka tako, da uporabnik z neko akcijo sproži izvajanje asinhronega zahtevka. Na odjemalcu se tvori objekt *XMLHttpRequest*, ki shrani funkcijo, ki se bo izvedla pri vračanju rezultata s strani strežnika (angl. callback function - povratna funkcija). Nato se določi naslov zahtevka, ki vsebuje programsko kodo na strani strežnika, zraven pa se posreduje tudi vrsta HTTP metode. Na koncu se pošlje celoten objekt, ki predstavlja zahtevek. Pri obdelavi zahtevka se na strani strežnika izvede potrebna koda, ki vrne rezultat. Preden se rezultat obdela na strani odjemalca se preveri, če se je zahtevek uspešno izvedel, nakar se prejeti rezultat ustrezno prikaže v HTML dokumentu. S tem je postopek končan, popravljen HTML dokument pa je viden pri uporabniku v brskalniku.

2.2 Tehnologije na strani strežnika

Sedaj pa bomo opisali tehnologije, ki smo jih uporabili pri razvoju aplikacije na strani strežnika.

2.2.1 Ruby

Ruby [20] je dinamičen, objektno usmerjen odprtokodni programski jezik. Objektno usmerjen pomeni, da temelji na objektih znotraj katerih se na-

hajajo podatki. Je močan, flekisibilen in enostaven za uporabo, aktualen pa je predvsem kot programski jezik spletnih aplikacij in se izvaja na strani strežnika. Po sintaksi je precej podoben Pythonu. Prav tako kot Python je Ruby interpretiran in ne prevajan. Ena izmed koristnih funkcionalnosti je tudi interaktivni interpreter imenovan irb, ki je uporaben predvsem za začetnike, saj omogoča interaktivno testiranje kode.

Jezik je razvil japonski programer Yukihiro Matsumoto (Matz) leta 1993 z namenom implementacije enostavnega skriptnega jezika po principu objektnega programiranja. Na razvoj jezika Ruby pa so vplivali jeziki kot so Perl, Pyhton in Lisp.

Ruby podpira več številčnih tipov [7], nize, polja in asociativna polja (angl. hash). Izpis podatka v neki spremenljivki dosežemo s postopkom imenovanim interpolacija (*"Pozdravljen #ime"*), določeni operatorji se lahko uporabljajo bodisi na številčnih tipih bodisi na nizih (+ kot seštevanje ali konkatenacija). Poleg klasičnih pogojnih in zančnih ukazov, kot so *if*, *else*, *switch*, *for* in *while* pozna Ruby tudi nekaj svojih, na primer *unless*, *loop*, *until* in tako naprej. Pri poljih je zanimivo, da lahko hranijo različne podatkovne tipe, posebnost pa je tudi bločna koda, ki se uporablja za iteriranje po elementih. Metode, ki imajo na koncu vprašaj (*veljaven?*) vračajo logični tip boolean, torej true ali false, zelo uporabni pa so simboli (*uporabnik[:ime]*) še posebaj v povezavi z asociativnimi polji.

2.2.2 Ruby on Rails

Ruby on Rails (ROR) [21] je ogrodje (angl. framework) namenjeno razvoju spletnih aplikacij v programskem jeziku Ruby. Ogrodje deluje po principu MVC (angl. Model-View-Controller), uporablja pa REST (angl. Representational State Transfer) arhitekturo. Pod ogrodje spada tudi ORM sistem ActiveRecord, ki skrbi za dostop do podatkovne baze. Prednost ORM-ja je varnost, saj programerju ni treba v nizih sestavljati SQL poizvedb nad bazo. Takšen način programiranja dostopa do podatkovne baze je zelo nevaren, saj lahko napadalec izvede napad imenovan SQL injekcija (angl. SQL injection).

Pri tem lahko zgolj s poznavanjem uporabniškega imena pridobi dostop do podatkov, torej brez gesla. To stori tako, da v SQL poizvedbo vrine logični ukaz (operator *ali*) in vrednost *true*. S tem nato izrabi pravilo Boolove algebre, kjer pri operatorju *ali* velja, da je izraz resničen, če je resničen vsaj eden izmed pogojev. Zato je potrebna še vrednost *true*.

Razvoj aplikacij v tem ogrodju zagovarja načelo agilnega razvoja, načelo kratke in neponavljajoče kode (angl. DRY - don't repeat yourself) ter načelo privzetih nastavitev (angl. convention over configuration).

Preden se lotimo programiranja, je potrebno nastaviti posebno datoteko imenovano Gemfile, v kateri se nahajajo knjižnice (Ruby gem-i) s katerimi definiramo odvisnosti znotraj aplikacije, ki so potrebne za delovanje.

Osnovni ukazi, ki se uporabljajo pri razvoju so:

- *rails new* - ustvari osnovno direktorijsko strukturo aplikacije
- *rails generate model* - ustvari nov model, tabelo v bazi s podanimi atributi in tipi
- *rails generate controller* - ustvari nadzornika s podanimi akcijami in HTML datotekami
- *rails console* - zagon konzole, testiranje
- *rails server* - zagon strežnika, kjer lahko privzeto dostopamo do aplikacije na naslovu localhost:3000
- *bundle exec rake db:reset* - ponastavitev podatkovne baze
- *bundle exec rake db:migrate* - poselitev, posodobitev podatkovne baze
- *bundle exec rake db:seed* - zapis podatkov v podatkovno bazo

2.2.3 Orodje ImageMagick

ImageMagick je programsko orodje namenjeno ustvarjanju, urejanju in pretvarjanju slik. Podpira več slikovnih formatov od JPG, GIF, PNG do TIFF,

PDF in SVG formata. Uporaba je možna v obliki CLI-ja (angl. command line interface), kar včasih lahko predstavlja slabost v primerjavi s programi z grafičnim vmesnikom kot sta Adobe Photoshop ali Gimp, vendar je tak način uporabe zelo primeren, če želimo uporabljati funkcionalnosti programa preko skripte v spletni aplikaciji.

Podpira več funkcij: animacijo, kompozicijo, upravljanje barv, DFT (diskretna Fourierjeva transformacija), risanje, enkripcijo, HDR (angl. high dynamic range), montažo, dodajanje teksta, transformacijo in razne posebne slikovne učinke.

Ena izmed teh funkcij oziroma ukazov je *mogrify*, na podlagi katere deluje naša spletna aplikacija za urejanje slik. S tem ukazom lahko enostavno dosežemo želeni slikovni učinek, kot je na primer sprememba velikosti podane slike v parametru (*mogrify -resize 50% slika.jpg*). Drugače pa naša aplikacija izrablja ImageMagick ukaze s pomočjo Ruby gema mini_magick [18].

2.3 Podporne tehnologije

Poleg prej predstavljenih tehnologij pa smo uporabili še nekaj tehnologij, ki so nam olajšale razvoj aplikacije.

2.3.1 Git

Git je sistem, ki se uporablja za verzioniranje programske kode. Predvsem pride prav, če neko zadevo razvija večja skupina ljudi. V tem primeru lahko izmenjevanje datotek poteka enostavneje, izboljša pa se tudi sledljivost napak. Git spada med porazdeljene sisteme za upravljanje verzij. Glavna komponenta je hramba oziroma drugače rečeno, je repozitorij izvorne kode.

Projekt pri katerem se uporablja Git, je razdeljen na tri dele: Delovni imenik, pripravljeno območje in imenik Git. Dostop do repozitorija lahko poteka preko različnih protokolov, bodisi git, https ali ssh.

Osnovni ukazi za delo so:

- *git add* - dodajanje novih datotek
- *git commit* - trajno dodajanje datotek v pripravljeno območje
- *git push* - nalaganje sprememb na oddaljeni repozitorij
- *git pull* - posodobitev lokalnih datotek s spremembami na oddaljenem repozitoriju

2.3.2 Heroku

Spletno aplikacijo smo objavili na oblačni platformi imenovani Heroku [12], ki podpira več programskih jezikov, predvsem pa je namenjena aplikacijam razvitim v ogrodju Ruby on Rails. Heroku je oblak tipa PaaS (angl. Platform as a Service), kar pomeni, da omogoča oblačne računske storitve namenjene razvoju, objavi in upravljanju aplikacij. Poleg tega lahko uporabnik kot razvijalec neke aplikacije hrani podatke potrebne za delovanje v podatkovni bazi PostgreSQL.

Sama objava aplikacije na platofrmi Heroku je dokaj enostavna. Najprej se je potrebno prijaviti na uradni spletni strani Heroku, nato na naš računalnik naložimo Heroku Toolbelt, ki v obliki ukazne vrstice našega operacijskega sistema omogoča poganjanje ukazov v obliki *heroku "ukaz"*. Preko ukazne vrstice se potem lahko prijavimo s Heroku računom. Preden poženemo ukaz *heroku create*, ki postavi osnovo za objavo, se moramo postaviti v direktorij, v katerem se nahaja naša aplikacija. Nato je potrebno samo še v git remote aplikacije dodati naš Heroku in zadeva je končana. Z ukazi tipa *git remote* upravljamo s skupino oddaljenih repozitorijev, lahko jih dodajamo, preimenujemo in podobno. Namreč na Heroku računu imamo dodaten repozitorij namenjen hrambi programske kode objavljene aplikacije. Posodabljanje objav poteka z ukazom *git push heroku*.

2.3.3 Amazon Web Service S3

Za trajno shranjevanje slik, ki jih nalagajo uporabniki, smo se odločili za storitev AWS (angl. Amazon Web Service) S3 (angl. Simple Storage Service) [2], pri kateri lahko shranjujemo datoteke v oblak.

Najprej je bilo potrebno ustvariti AWS račun, kjer je potrebno navesti tudi podatke o kreditni kartici, saj je storitev zastonj le prvo leto, nato pa je potrebno plačati glede na zasedenost oziroma potrebe. Kasneje je bilo treba še ustvariti uporabnika preko AWS Identity and Access Management (IAM), ki mu omogočamo uporabo storitve in dostop do 5GB oblačnega prostora. S tem se ustvarita dostopni in skriti ključ, ki predstavljata neko vrste avtorizacije, kdo oziroma katera aplikacija lahko uporablja to storitev. Zadnji korak je postavitev prostora imenovanega "bucket".

S tem je konfiguracija končana, sledi le še zapis podatkov kot so dostopni ključ, skriti ključ in "bucket"v aplikacijo na platformi Heroku. To storimo z ukazom *heroku config:set*, da nam ni treba zapisati zadev neposredno v programsko kodo aplikacije.

2.3.4 Android - Java in JSON

Razvoj mobilne aplikacije je potekal v programskem jeziku Java, ki je namenjena uporabnikom mobilnih naprav z operacijskim sistemom Android. Java je objektno usmerjen programski jezik, ki je statično tipiziran, kar pomeni, da moramo pred inicializacijo določiti tip spremenljivke.

Programiranje Android aplikacij temelji na razredih *Activity*, ki predstavljajo nek zaslon, ki je viden v aplikaciji. Primer 2.4 prikazuje primer zaslona oziroma aktivnosti. Znotraj posamezne aktivnosti se vedno kliče metoda onCreate(), ki definira glavno metodo, namenjeno prikazu trenutne aktivnosti oziroma zaslona. Željeno postavitev podob dosežemo preko XML datotek s pomočjo razvrščevalnikov (angl. layout), ki omogočajo različne postavitve podob v grafičnem vmesniku aplikacije. Poleg tega pa je potrebno v datoteko AndroidManifest.xml zapisati vsa dovoljenja, ki jih potrebuje aplikacija,

bodisi je to na primer dostop do interneta ali kamere.

Primer 2.4: Primer Java programske kode za Android

```
public class MainActivity extends AppCompatActivity {
   Button btn;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      btn=(Button)findViewById(R.id.btn);
      btn.setOnClickListener(new OnClickListener() {
         public void onClick(View v){
            //koda
         }
      }
   }
}
```

Ker je za delovanje aplikacije potrebna prijava, predstavlja del mobilne aplikacije tudi spletna aplikacija, ki ob prijavi vrača odgovore v formatu oblike JSON-a (angl. JavaScript Object Notation). JSON je programsko neodvisen format, kjer so podatki zapisani v obliki asociativnih polj ({ *"ključ": "vrednost"* }).

Poglavje 3

Podatkovni model

Delovanje spletne in mobilne aplikacije sloni na precej enostavnem podatkovnem modelu. Entiteno-realicijski diagram je sestavljen zgolj iz dveh entitet oziroma tabel. Prva entiteta hrani podatke o uporabniku, ki je povezana na drugo eniteto, kjer se hranijo podatki o sliki. Povezava med njima ima kardinalnost ena proti mnogo (1:n), kar pomeni, da ima uporabnik lahko več slik, ena slika pa pripada natanko enemu uporabniku.

Aplikacija, ki je objavljena na platformi Heroku ima pripadajočo podatkovno bazo prav tako shranjeno na isti platformi, kjer podpirajo sistem PostgreSQL. PostgreSQL je odprtokodni objektno-relacijski podatkovni sistem namenjen varnem hranjenju podatkov. Temelji na standardu ACID (angl. Atomicity, Consistency, Isolation, Durability). Atomarnost pomeni, da omogoča transakcije. V primeru neuspeha izvajanja nekega ukaza v zaporedju ukazov se ostali ukazi tega zaporedja razveljavijo. Konsistenca zagotavlja ustreznost zapisanih podatkov na podlagi pravil in omejitev. Izolacija omogoča, da vzporedne izvedene transakcije dajo enako stanje, kot če bi bile izvedene zaporedno. Zanesljivost pa zagotavlja, da ko so podatki enkrat zapisani, tudi ostanejo zapisani.

Za hrambo slik pa se uporablja oblačna storitev AWS S3, ki posledično razbremeni delovanje podatkovne baze, saj se vanjo zapiše le naslov URL (angl. Uniform Resource Locator) na katerem je dostopna slika, namesto

celotne slike. V nasprotnem primeru bi bilo treba sliko zakodirati v format Base64, da bi se lahko zapisala v podatkovno bazo v obliki niza znakov, za prikaz pa bi bilo tako potrebno še dekodiranje.

3.1 Načrtovanje podatkovnega modela

Imamo enostaven podatkovni model, kjer imamo dve entiteti povezani z eno povezavo. Eniteta User (uporabnik) hrani podatke o uporabniku in je povezana na entito Image (slika), ki hrani podatke o sliki. Povezava med njima ima kardinalnost ena proti mnogo (1:n). Na sliki 3.1 je prikazan model podatkovne baze zgrajen v programskem orodju MySQL Workbench.

Slika 3.1: Entitetno relacijski model podatkovne baze, ki se uporablja pri razviti alikaciji.

3.1.1 Struktura entitete User - Uporabnik

Entiteta User je osnovna struktura, ki hrani podatke o uporabniku. Sestavljena je iz 9 atributov. Primarni ključ, ki ga že ustvari Ruby-jeva komponenta za ORM ActiveRecord, je pri generiranju modela *id*, enolični identi-

fikator, ki je tipa INT (Integer - celoštevilski tip). Za prijavo uporabnika je potreben veljaven e-naslov, ki se shrani v atribut *email* in uporabniško ime, ki se shrani v atribut *username*. Tukaj mora uporabnik vnesti veljavno uporabniško ime, ki vsebuje od štiri do šestnajst znakov. Za konec pa je potrebno še geslo, atribut *password_digest*, ki mora biti izbrano v skladu s pravili, v primeru naše aplikacije je to od šest do osemnajst znakov, med katerimi mora biti vsaj ena številka. V bazo se ne shrani čistopis, ampak se shrani zgoščena vrednost.

Zraven se shrani tudi status, ki definira ali je uporabnikov računov aktiviran ali deaktiviran, zato je tipa CHAR (Character - znak), pri čemer se shrani A kot aktiviran ali D kot deaktiviran. Administratorja smo kreirali ročno, tako da smo v atribut *admin*, ki je tipa BOOLEAN, zapisali vrednost true, ostalim, ki se prijavljajo, pa se avtomatsko zapiše false.

Uporabnik ima ob vpisu tudi možnost ustvarjanja piškotkov, torej možnost "remember me", kjer se ob tem v atribut *remember_digest* zapiše podobna zadeva kot pri atributu *password_digest*. Ruby-jeva komponenta ORM ob kreiranju novega uporabnika zraven še ustvari dva atributa, kamor se zapišeta datuma ustvarjanja in posodabljanja konkretnega uporabnika (*created_at* in *updated_at*).

3.1.2 Struktura entitete Image - Slika

Do te entitete lahko dostopa le uporabnik, ki je prijavljen, torej že ima ustvarjeno entiteto User. Ta entiteta je namenjena shranjevanju podatkov o sliki, ki jo uporabnik naloži v aplikacijo. Podobno kot prejšnja entiteta ima tudi ta entiteta atribut *id*, enolični identifikator v obliki cele pozitivne številke. Zraven ima tudi atribut *name*, ki predstavlja ime slike, in hrani niz znakov maksimalne dolžine šestnajst (VARCHAR(16)). Pri nalaganju slike se mora uporabnik tudi odločiti ali želi sliko objaviti, s čimer določi javni dostop. S tem se v atribut tipa BOOLEAN z imenom *private* shrani vrednost false. Če pa želi imeti zaseben dostop do slik, pa se zapiše vrednost true, kar pomeni, da je slika zasebna.

Najpomembnejši atribut v tej entiteti pa je *source*, kamor se zapiše URL naslov slike, ki je shranjena v storitvi AWS S3. Ta atribut je tipa VARCHAR maksimalne dolžine 256 znakov. Poleg tega ima še eno dodatno omejitev, namreč sprejemajo se le nekateri slikovni formati, JPG oziroma JPEG, PNG in GIF. Komponenta za ORM v ogrodju Rails tudi tukaj kreira dva dodatna atributa *created_at* in *updated_at*. V tej entiteti se nahaja tudi tuji ključ, imenovan *user_id*, ki beleži kateremu uporabniku pripada slika.

3.2 Uporabniške vloge

Spletna aplikacija ponuja tri uporabniške vloge. Do dela funkcionalnosti aplikacije lahko dostopajo vsi, tudi tisti, ki niso registrirani. Pri tem lahko lokalno urejajo slike, vendar imajo na voljo le peščico slikovnih učinkov.

Registriranim uporabnikom je omogočena popolna funkcionalnost. To pomeni, da lahko nalagajo in hranijo slike na svojem računu, izbirajo dostopnost do slik, jih lahko delijo in najpomembnejše, urejajo na kopico možnih načinov. Sem spada še tretja uporabniška vloga administrator (*admin=true*), ki ima dodatne pravice, s čimer lahko nadzira spletno aplikacijo in posledično briše uporabnike.

Mobilna aplikacija pa je namenjena zgolj registriranim uporabnikom. Pri tem je najprej potrebna avtentikacija, šele nato se lahko posneto fotografijo naloži na določen uporabniški račun.

Poglavje 4

Razvoj aplikacije

Razvoj aplikacije je potekal s pomočjo več orodij oziroma programske opreme. Razvoj spletne aplikacije v ogrodju Ruby on Rails je potekal v razvojnem okolju JetBrains IntelliJ IDEA [14]. Za poganjanje raznih ukazov tipa *rails*, *heroku* in *git* smo uporabljali GitBash [9]. Mobilna aplikacija pa je bila razvita v razvojnem okolju Android Studio [4].

Spletna aplikacija ima kot projekt naslednjo strukturo, našteti so le pomembnejši direktoriji:

- *app* - jedro aplikacije, programske kode (pogledi, modeli, nadzorniki)
- *app/assets* - datoteke, ki skrbijo za izgled aplikacije, ter statične datoteke kot so slike
- *bin* - zagonske datoteke
- *config* - datoteke za konfiguracijo, na primer config.ru
- *db* - datoteke potrebne za delovanje podatkovne baze
- *log* - datoteke za beleženje dogodkov v aplikaciji
- *public* - javne datoteke, ki so dostopne vsem
- *tmp* - začasne datoteke
- *vendor* - programska koda gem-ov

4.1 Odjemalec

Pri razvoju aplikacije na strani odjemalca smo se predvsem ukvarjali s strukturiranjem ter prikazom podatkov in izdelavo grafičnega vmesnika. Uporabljali smo ogrodje Bootstrap [6], ki je namenjeno izdelavi odzivnih spletnih aplikacij, oziroma aplikacij, ki se prilagajajo velikostim zaslona različnih naprav. Za uporabo je potrebno naložiti datoteke Bootstrap tipa CSS in JS, pri delovanju pa se uporablja jQuery, zato je potrebno naložiti tudi to knjižnico.

Izdelava grafičnega vmesnika v ogrodju Bootstrap temelji predvsem na podlagi vnaprej definirah razredov, ki jih dodamo našim HTML elementom. Osnovna struktura prikazane vrstice se lahko razbije na mrežo z dvanajstimi polji. V primeru 4.1 je prikaz razdelitve vrstice na štiri enake dele (3+3+3+3=12).

Primer 4.1: Vrstica s štirimi stolpci

```
<div class="row">
   <div class="col-md-3"></div>
   <div class="col-md-3"></div>
   <div class="col-md-3"></div>
   <div class="col-md-3"></div>
</div>
```

Možne so tudi razne kombinacije širine vrstice in zamika, kot je delo s samo polovico vrstice, ki je od roba zamaknjena za tretjino zaslona, kar je prikazano na primeru 4.2.

Primer 4.2: Zamik vsebine za tretjino zaslona

```
<div class="row">
   <div class="col-md-6 col-md-offset-3"></div>
</div>
```

Uporaba ogrodja Bootstrap pride prav tudi pri oblikovanju raznih obrazcev oziroma form. V ta namen imamo posebne razrede, s katerimi lahko definiramo izpis vnosnega polja v svoji vrstici. Lahko pa se vsa polja izpišejo

v isti vrstici, kot je prikazano v primeru 4.3.

Primer 4.3: Obrazec za vpis podatkov

```
<form class="form-inline">
   <div class="form-group">
      <label class="sr-only" for="exampleInputEmail">Email
          address</label>
      <input type="email" class="form-control"
          id="exampleInputEmail" placeholder="Email">
   </div>
   <div class="form-group">
      <label class="sr-only"
          for="exampleInputPassword">Password</label>
      <input type="password" class="form-control"
          id="exampleInputPassword" placeholder="Password">
   </div>
   <div class="checkbox">
      <label>
         <input type="checkbox"> Remember me
      </label>
   </div>
   <button type="submit" class="btn btn-default">Sign in</button>
</form>
```

Prav tako lahko že napisane razrede, ki jih uporablja ogrodje Bootstrap dodelamo po svojih željah. Če na primer ne želimo klasičnega gumba, lahko uporabimo Bootstrap gumb, ki ga potem lahko še naknadno grafično dodelamo. Za to lahko v naši CSS datoteki ponovno definiramo želen razred in ga dopolnimo s svojimi lastnostmi.

Namesto za razne gumbe, v katere bi bilo pisanje besedila preveč potratno, smo se rajši odločili kar za uporabo ikon imenovanih Material Icons [11]. Osnovnemu ogrodju Bootstrap smo dodali tudi razširitev imenovano Jasny Bootstrap [16], ki smo jo uporabili pri funkcionalnosti za nalaganje

slik na uporabniški račun, kjer se v polje naloži predogled (angl. preview) slike, preden se ta naloži, kar je prikazano v primeru 4.4.

Primer 4.4: Predogled slike

```
<div class="fileinput fileinput-new" data-provides="fileinput">
   <div class="fileinput-preview thumbnail"
       data-trigger="fileinput" style="width: 200px; height:
       150px;"></div>
   <div>
   <span class="btn btn-default btn-file"><span
       class="fileinput-new">Select image</span><span
       class="fileinput-exists">Change</span><input type="file"
       name="image"></span>
   <a href="#" class="btn btn-default fileinput-exists"
       data-dismiss="fileinput">Remove</a>
   </div>
</div>
```

Za popolno delovanje AJAX-a je bilo potrebno del sprogramirati tudi v JavaScript-u. Odločili smo se za uporabo knjižnice jQuery, ki malce pospeši in olajša delo. Primer programske kode, ki sestavi AJAX zahtevek je prikazan v primeru 4.5.

Primer 4.5: Sestavljanje AJAX zahtevka

```
$(function () {
   $('#sepia').on('click', function () {
   var s = $("#source").val();
   sepia(s);
   });
});
function sepia(source) {
   $.post('/sepia.json', {
      imagesource: source
   }, function (data) {
```

```
      var output = data.result;
   });
   var div = $('#target').attr('src', source + '?' + new
       Date().getTime());
   $('#image_plate').html(div);
}
```

Za urejanje slik na strani odjemalca smo uporabili knjižnico glfx [10]. Uveljavljanje raznih slikovnih učinkov poteka s pomočjo tehnologije WebGL.

V primeru 4.6 je prikazano, kako se najprej ustvari *canvas* HTML element (risalna površina). Nahaja se v try-catch bloku, saj je WebGL dokaj nova tehnologija in se lahko zgodi, da ni podprta. Nato je potrebno sliko, ki se nahaja v img HTML elementu z identifikatorjem *image_show* pretvoriti v teksturo (angl. texture). Ta struktura predstavlja dejansko sliko oziroma podatke o posameznem slikovnem elementu. Pri urejanju nad canvas elementom kličemo več funkcij (angl. chaining). Prva izmed njih je *draw*, ki preslika podatke oziroma teksturo v *canvas*. Po tem se kliče funkcija želenega slikovnega učinka, na primer svetlost (*brightnessF()*). Zato da je rezultat viden, se na koncu uporabi funkcija *update()*. Da je po urejanju uporabniku na voljo prenos urejene slike, se iz canvas HTML elementa znova ustvari img HTML element, kar storimo s funkcijo *toDataURL()*.

Primer 4.6: Uporaba glfx knjižnice

```
var canvas; //INIT
try {
   canvas = fx.canvas();
} catch (e) {
   alert(e);
   return;
}
function brightnessF(v) {
   var image = document.getElementById('image_show');
   var texture = canvas.texture(image);
```

```
    canvas.draw(texture).brightnessContrast(v, 0).update();
    image.src = canvas.toDataURL('image/png');
    $('#cbdownload').attr('href', image.src);
    $('#cbdownload').attr('download', "file.png");
}
```

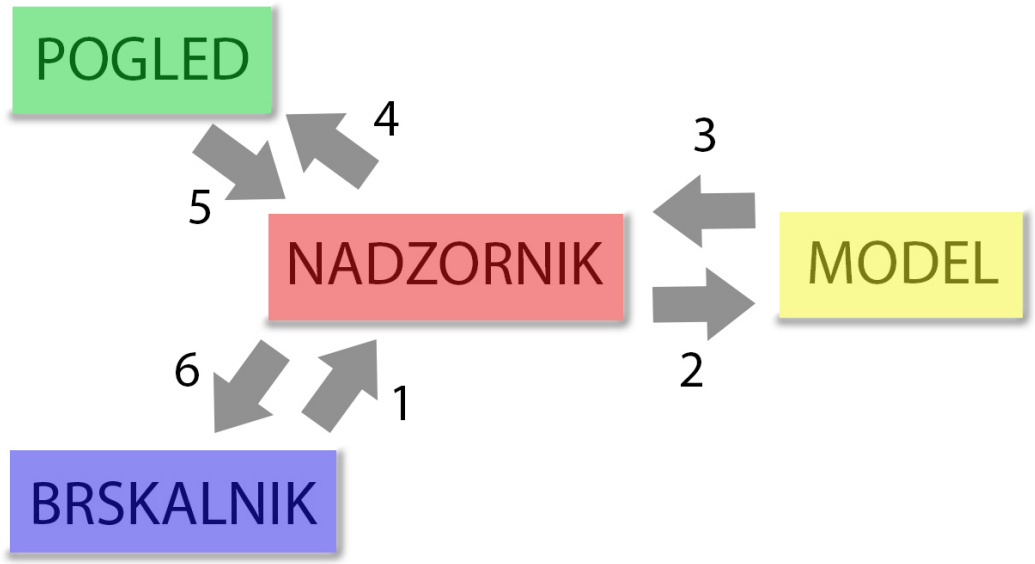
4.2 Strežnik

Razvoj spletne aplikacije na strani strežnika smo opravili v programskem jeziku Ruby. Pri tem smo uporabili ogrodje Ruby on Rails (ROR), ki temelji na principu MVC. MVC je arhitekturni vzorec, ki se uporablja pri implementaciji aplikacij z grafičnim uporabniškim vmesnikom ter tudi spletnih aplikacijah. Sestavljen je iz treh komponent in sicer modela (angl. model), pogleda (angl. view) in nadzornika (angl. controller). Vsaka izmed njih ima svojo nalogo pri delovanju. Model se nanaša na entitete in povezave med njimi v podatkovni bazi. Pogled je namenjen prikazu podatkov, ki so mu pri komunikaciji posredovani. Nadzornik pa na podlagi uporabnikovih akcij narekuje delovanje, torej katere podatke uporabnik zahteva, kje in kako bodo prikazani in tako naprej.

Če natančneje pregledamo interakcijo med komponentami, ki so prikazane na sliki 4.1, vidimo, da uporabnik z neko akcijo, v aplikaciji sproži dogodek, ki se posreduje nadzorniku (1). Ta na podlagi tega določi, katere podatke mora pridobiti iz podatkovne baze. Za podatke zaprosi model (2), ki mu jih posreduje. Ko nadzornik pridobi želene podatke (3), jih posreduje pogledu (4), ta sestavi ustrezen HTML dokument, ki ga vrne nadzorniku (5), ta ga pa kot odgovor na zahtevo prikaže uporabniku (6).

4.2.1 Model

Model v MVC arhitekturi je namenjen komunikaciji s podatkovnim modelom oziroma podatkovno bazo, ki je predstavljena v obliki entitet (tabel) z določenimi atributi in njihovimi omejitvami, kot je na primer podatkovni tip.



Slika 4.1: Grafični prikaz delovanja MVC

Pri razvoju smo uporabili Ruby-jevo komponento ORM imenovano ActiveRecord.

S pomočjo ORM-ja lahko entiteto v podatkovni bazi predstavimo, kar kot razred, ki razširja temu namenjen Ruby-jev razred *ActiveRecord::Base*. Lahko rečemo, da iz entitetno-relacijskega pogleda na podatkovno bazo ustvarimo objektni pogled na podatkovno bazo. Nato lahko v tem razredu atribute, ki jih vsebuje ta entiteta, zapišemo kot atribute razreda v obliki spremenljivk s podanimi omejitvami. Lahko jim določimo podatkovni tip (niz, celo število, logični tip), obveznost, edinstvenost (angl. uniqueness - pojavi se le enkrat), format zapisan z regularnim izrazom in tako naprej. Omejitve določimo z ukazom *validates*, ki poleg tega poskrbi še za validacijo na strani strežnika.

Validacija vnosa je potrebna, da uporabniku sporočimo pravilnost vnosa podatkov, ki naj bi bili zapisani v podatkovno bazo. Na primer, če želimo vpis tromestnega števila, moramo v primeru vnosa preveč ali premalo števil ali morda celo vnosa znakov, ki niso numerični, sporočiti napako. Poznamo

validacijo na strani odjemalca, ki jo omogoča HTML5. Prednost tega pristopa je, da se preverjanje izvede še preden se zahtevek pošlje na strežnik. S tem ni obremenjen strežnik, poleg tega pa se preverjanje izvede hitreje. Vendar to lahko bolj izkušen uporabnik obide in sam spiše zahtevek na strežnik, zaradi česar pa je potrebna še validacija na strani strežnika, preden se podatki dejansko zapišejo. Tudi pri tem načinu je treba v primeru napake sporočiti, da je potreben ponoven vnos. Potrebno je tudi sporočiti, katera polja so napačna in kakšen vnos zahtevajo.

Ker naš podatkovni model ni sestavljen le iz ene entitete, je v modelu potrebno definirati tudi povezave. V našem primeru imamo entiteto User, ki je povezana na entiteto Image. Torej določen uporabnik ima lahko več slik, vendar določena slika pripada natanko enemu uporabniku. Zato pri povezavi s strani entitete User uporabimo ukaz *has_many* (ima več). S simbolom, ki sledi ukazu, povemo, za katero entiteto gre. Na strani entitete Image, uporabimo ukaz *belongs_to* (pripada) in simbol, ki definirana entiteto, na katero je ta povezana.

Če želimo uporabljati nek atribut, do katerega uporabnik ne dostopa in ga ne spreminja, torej ni potrebna validacija, uporabimo ukaz *attr_accesor*. S tem lahko potem le mi, razvijalci, upravljamo v ozadju delovanja aplikacije. Obstaja tudi posebna metoda imenovana *before_save*, ki se uporablja v primeru spreminjanja uporabnikovega vnosa pred zapisom v podatkovno bazo. To smo na primer uporabili pri kreiranju uporabnika. Preden se shrani se vedno nastavi atribut *status* na A (aktiviran). Primer 4.7 prikazuje razred, ki predstavlja uporabnika (entiteta User).

Primer 4.7: Razred entitete User

```
class User < ActiveRecord::Base

  has_many :images, dependent: :destroy

  before_save { self.email = email.downcase }
  before_save { self.status = 'A' }
```

```
VALID_USERNAME_REGEX = /\A[a-z0-9\-_].{3,16}\z/i
validates :username, presence: true, length: { minimum: 4,
    maximum: 16 },
format: { with: VALID_USERNAME_REGEX }, uniqueness: {
    case_sensitive: false }

VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
validates :email, presence: true, length: { maximum: 32 },
format: { with: VALID_EMAIL_REGEX },
uniqueness: { case_sensitive: false }

has_secure_password
VALID_PASSWORD_REGEX = /\A(?=.*[a-z])(?=.*[0-9]).{5,18}\z/i
validates :password, presence: true, length: { minimum: 6,
    maximum: 18 }, format: { with: VALID_PASSWORD_REGEX }

attr_accessor :remember_token
```

V primeru 4.7 smo implemetirali entiteto User s pomočjo ORM-ja. Določen uporabnik ima lahko več slik, zato smo uporabili ukaz *has_many :images* Dodaten ukaz *dependent: :destroy* pomeni, da se, ko izbrišemo uporabnika, izbrišejo tudi njegove slike. Uporabili smo tudi več metod, ki se izvedejo pred shranjevanjem v podatkovno bazo (*before_save*). Pri vnosu uporabniškega imena je zahtevan vnos (angl. required), pri katerem mora biti dolžina med štirimi in šestnajstimi znaki. Poleg tega pa tudi zahtevamo niz oblike, ki zadošča regularnemu izrazu. Podobno strukturo imamo tudi pri elektronskem naslovu.

Pri razvoju aplikacije smo prav tako poskrbeli za varnost naših uporabnikov. Pri shranjevanju gesla v podatkovno bazo ne zapisujemo čistopisa, ampak uporabimo Ruby gem imenovan Bycrpt, ki je namenjen kriptiranju gesel. Pri atributu *password_digest* v podatkovni bazi uporabimo ukaz *has_secure_password*, ki na podlagi vnesenega niza ustvari nov, na videz na-

ključen, niz znakov. Iz tega novo nastalega niza (tajnopis) je praktično nemogoče pridobiti čistopis gesla. Seveda so razvijalci ogrodja implemetirali tudi metodo *authenticate*. S to metodo lahko primerjamo enakost čistopisa in tajnopisa in posledično avtenticiramo uporabnika. Pri izbiri gesla smo nastavilli tudi določene omejitve. Pravilno geslo vsebuje od pet do osemnajst znakov, pri čemer mora biti vsaj en numeričen znak.

4.2.2 Nadzornik

Naloga nadzornika je, da upravlja delovanje spletne aplikacije. Torej spremlja uporabnikove akcije in na podlagi njih delegira delovanje modela ter pogleda. Osnovno strukturo nadzornikov in akcij imamo v ogrodju ROR zapisano v datoteki *routes.rb*. V tej datoteki imamo definirane posamezne nadzornike, akcije nad njimi in katero HTTP metodo podpira posamezna akcija. V primeru 4.8 imamo prikazan del te datoteke. Prva stran, ki se vedno prikaže pri vstopu je glavna stran, na podlagi programske kode, ki je zapisana v akciji *home*. Pri pregledovanju galerije se pošlje GET zahtevek na naslov /gallery, s tem se kliče akcija *gallery* v nadzorniku *StaticController* (*static#gallery*) . Podobno je tudi pri ostalih nadzornikih, prikazanih v primeru 4.8.

Primer 4.8: Datoteka routes.rb

```
root 'static#home' #/
get 'gallery' => 'static#gallery' #/gallery
get 'help' => 'static#help' #/help
get 'upload' => 'static#upload' #/upoad
get 'users/new'
get 'signup' => 'users#new'
get 'settings' => 'users#settings'
patch 'reset' => 'users#reset'
resources :users
```

Posebnost je ukaz *resources*, ki definira arhitekturni stil imenovan Representational State Transfer (REST). Značilnost REST-a je, da poteka komu-

nikacija tako, da si odjemalec in strežnik izmenjujeta reprezentacijo virov. Torej pravilen naslov in pravilna HTTP metoda sprožita ustrezno akcijo, katera opravi neko delo in nato sporoči odgovor. Pri uporabi ukaza *resources :users* se ustvari struktura delovanja aplikacije na viru /users.

Osnovni viri in podviri so:

- GET /users - pridobi vse uporabnike
- GET /users/1 - pridobi uporabnika z id-jem 1
- GET /users/new - prikaz forme za kreiranje računa
- POST /users - zapis novega uporabnika v podatkovno bazo
- DELETE /users/1 - zbriše uporabnika z id-jem 1
- GET /users/1/edit - forma za posodobitev uporabnika z id-jem 1
- PATCH /users/1 - posodobitev uporabnika z id-jem 1, zapis v podatkovno bazo

Programiranje nadzornika poteka preko razreda, ki razširja razred *ApplicationController*. To je glavni nadzornik, ki se ustvari z aplikacijo, razširjen je iz razreda *ActionController::Base*. Neko akcijo zapišemo kot metodo, ki se ob dostopu do nekega naslova sproži. Z ukazom *before_action* lahko pred začetkom izvajanja akcije sprožimo določene metode. Na primer, preden si lahko določen uporabnik ogleda profil nekoga drugega (akcija *show*) se sproži metoda *logged_in_user*, ki preveri, ali je uporabnik trenutno vpisan v aplikacijo, oziroma ali je sploh registriran. Ostali (neregistrirani uporabniki) tega ne morejo. Če je vse v redu, se izvede programska koda, ki v spremenljivko *@user* (angl. instance variable - spremenljivka instance) shrani objekt oziroma vse podatke tega uporabnika in to posreduje pogledu, ki prikaže ustrezne podatke. Podobno stori tudi s spremenljivko *@images*, kamor se shrani seznam slik, ki pripradajo temu uporabniku. V primeru 4.9 je prikazan del nadzornika z osnovnimi akcijami, ki jih definira REST arhitektura.

Primer 4.9: Nadzornik za uporabnike

```
class UsersController < ApplicationController
   def index
      @users = User.all.paginate(page: params[:page], :per_page =>
         24)
   end

   def new
      @user = User.new
   end

   def show
      @user = User.find(params[:id])
      @images = @user.images.paginate(page: params[:page])
   end

   def create
      @user = User.new(user_params)
      if @user.save
         log_in @user
         redirect_to '/'
      else
         render 'new'
      end
   end

   def destroy
      User.find(params[:id]).destroy
      redirect_to users_url
   end

   # OTHER ACTIONS
end
```

V primeru kreiranja uporabniškega računa mora uporabnik dostopati do naslova /users/new, kjer se sproži akcija *new*, ki pripravi objekt za shranjevanje podatkov in prikaže formo. S klikom na gumb za ustvarjanje računa se pošlje POST HTTP zahtevek na naslov /users, kjer se izvede akcija *create*. Nato se ustvari User objekt s podanimi podatki. Pred ustvarjenjem novega objekta Ruby-jeva komponenta ORM preveri ustreznost podatkov in nato shrani podatke v podatkovno bazo. Pri tem se uporabljajo Ruby-jevi močni parameteri (angl. strong parameters). Ti so namenjeni določanju, katere atribute lahko uporabnik vpisuje oziroma manipulira. Če je vse v redu in se podatki uspešno shranijo v podatkovno bazo, se ustvari seja s funkcijo *log_in* in steče preusmeritev na začetno stran aplikacije. Sicer pa se ponovno naloži forma, ki vsebuje opozorilo o napakah.

HTTP je protokol, ki ne hrani stanja (angl. stateless). To pomeni, da se ob preusmeritvah ne hranijo podatki, ki so bili na prejšnji strani. Zato je potrebno ustvariti sejo, kar je prikazano v primeru 4.10. Seja omogoča hranjenje podatkov o tem, kdo je vpisan in podobno. Za to je potreben dodaten nadzornik imenovan *SessionsController*, ki ima dodatno še spisan modul, kjer so zapisane razne pomožne metode za delovanja nadzornika in celotne aplikacije.

Primer 4.10: Glavna metoda za ustvarjanje seje

```
def create
   user = User.find_by(username: params[:session][:username])
   if user && user.authenticate(params[:session][:password])
      log_in user
      remember user
      params[:session][:remember_me] == '1' ? remember(user) :
          forget(user)
      redirect_to ('/')
   else
      flash.now[:danger] = 'Invalid username/password combination!'
      render 'new'
```

```
    end
end
```

Pri vpisu v aplikacijo avtenticiramo uporabnika na podlagi vpisanega uporabniškega imena in gesla. Torej, če je registriran, najprej poiščemo uporabnika po uporabniškem imenu. Nato uporabimo metodo *autheticate*, s katero primerjamo vpisano geslo (čistopis) in tajnopis gesla ustvarjenega pri registraciji, ki je zapisan v podatkovni bazi. Če vpisano ustreza podatkom v podatkovni bazi, je vpis uspešen, s čimer se kliče pomožna metoda *log_in*, ki shrani *id* vpisanega uporabnika v polje *session[:user_id]*. V primeru izbrane funkcionalnosti "Remember me" se kliče tudi metoda *remember_me*, ki ustvari piškotek. Ta podatek, ki ga predstavlja piškotek, omogoča, da se ob naslednjem dostopu do aplikacije uporabniku ni potrebno avtenticirati, ampak se vpiše že avtomatsko. Nekatere pomožne metode, ki jih potrebujemo pri delu s sejami, so prikazane v primeru 4.11.

Primer 4.11: Pomožne metode pri seji

```
def log_in(user)
  session[:user_id] = user.id
end

def remember(user)
  user.remember
  cookies.permanent.signed[:user_id] = user.id
  cookies.permanent[:remember_token] = user.remember_token
end

def forget(user)
  user.forget
  cookies.delete(:user_id)
  cookies.delete(:remember_token)
end
```

Urejanje se prične s klikom na izbrano sliko. Na novi strani je na levi

strani prikazana slika, na desni pa se nahajajo gumbi namenjeni urejanju. S klikom na posamezen gumb se nad sliko prikaže obrazec. Ta obrazec je lahko v obliki posameznega gumba, drsnika ali vnosnega polja. S klikom na gumb obrazca se sproži urejanje, AJAX zahtevek poskrbi za pridobivanje parametrov iz obrazca in njihov nadaljnji prenos. Na podlagi gumba (*id*) zahtevek ugotovi za kateri slikovni učinek gre in nato sestavi POST zahtevek, kjer se izvede urejanje. Po končanem urejanju se kliče funkcija, ki poskrbi za ponovno nalaganje slike (osveževanje). Pri tem pa moramo počakati dve sekundi, da se prejšnja verzija zagotovo izbriše iz hrambe in nova uspešno naloži. Za to uporabimo funkcijo *setTimeout()*. Primer sestavljanja AJAX zahtevka za rezanje slike (angl. crop) je zapisan v primeru 4.12.

Primer 4.12: Sestavljanje AJAX zahtevka

```
jQuery.ajax({
   type: "POST",
   url: "/crop",
   data: {"coordinates": val, "imagesource": s},
   complete: function () {
      setTimeout(function(){
         var div = $('#target').attr('src', source + '?' + new
            Date().getTime());
         $('#image_plate').html(div);
      }, 2000);
   }
});
```

Urejanje nad sliko se izvede na strani strežnika in je prikazano v primeru 4.13. Za primer je prikazana akcija *crop*, ki se izvede ob POST zahtevku na naslovu /crop. Najprej se preberejo parametri iz zahtevka, ki ga je poslal AJAX. V parameter *coordinate* se zapiše seznam šestih parametrov. Drugi in tretji parameter predstavljata točko levega zgornjega kota, kjer se rezanje prične. Ničelni in prvi parameter sta potrebna za določitev preostalih treh kotov slike, drugače rečeno to sta širina in višina obrezanega dela slike. Četrti

parameter predstavlja razmerje med naravno (originalno) in prikazano širino slike. Peti parameter predstavlja enak koncept za višino. Ti razmerji sta pomembni, saj je potrebno normalizirati koordinate rezanja, saj je zaradi lažjega dela je pri urejanju slika pomanjšana, da je vidna v celoti.

Primer 4.13: Akcija za urejanje slike

```
def crop
   coordinate = params["coordinates"]
   imagesource = params["imagesource"].to_s
   wratio = coordinate[4].to_f
   hratio =coordinate[5].to_f

   w = coordinate[0].to_f * wratio
   h = coordinate[1].to_f * hratio
   x1 = coordinate[2].to_f * wratio
   y1 = coordinate[3].to_f * hratio

   image_object = MiniMagick::Image.open("#{imagesource}")
   image_object.combine_options do |i|
      i.crop "#{w}x#{h}+#{x1}+#{y1}!"
   end
   aws_update(imagesource, image_object)

   respond_to do |format|
    format.json { head :no_content }
   end
end
```

S klicem metode *MiniMagick::Image.open("#imagesource")* se na podlagi URL-ja odpre slika in shrani v objekt, ki ga kasneje lahko urejamo. V tem primeru nad danim objektom kličemo metodo *crop*, ki obreže sliko na podlagi parametrov. Za shranjevanje (posodabljanje) slike poskrbi metoda *aws_update*, ki je prikazana v primeru 4.14.

Primer 4.14: Metoda za posodabljanje slike v produkciji

```
def aws_update(imagesource, image_object)
   key = (imagesource.reverse!.split('/').first).reverse!
   extension = (key.reverse!.split('.').first).reverse!
   temp_file = Tempfile.new(["temp",".#{extension}"])
   temp_path = temp_file.path
   image_object.write "#{temp_path}"

   # AWS
   Aws.config.update({ region: 'eu-central-1',
      credentials: Aws::Credentials.new('#{access_key}',
         '#{secret_key}')
   })

   key = key.reverse!
   client = Aws::S3::Client.new(region: 'eu-central-1')
      resp1 = client.delete_object({ bucket: "imageplusplus", key:
         "uploads/#{key}", use_accelerate_endpoint: false })

   File.open("#{temp_path}", 'rb') do |file|
      resp2 = client.put_object(bucket: "imageplusplus", key:
         "uploads/#{key}", body: file, acl: "public-read-write")
   end
end
```

Metoda prikazana v primeru 4.14 prikazuje posodabljanje slike, po urejanju, v produkciji. Za to smo uporabili gem aws-sdk [5]. Urejeno sliko, trenutno shranjeno v objektu, zapišemo v začasno datoteko (*Tempfile*). Problem, na katerega smo naleteli je namreč, da je posodabljanje slike mogoče le, če je zapisana na disku. Za tem je potrebna avtentikacija za brisanje in shranjevanje na AWS storitev. Po avtentikaciji najprej pobrišemo prejšnjo verzijo slike (pred urejanjem). To storimo z metodo *delete_object*, kateri posredujemo za kateri "bucket"gre in za katero sliko gre. Njeno ime je namreč

njen enolični identifikator. Nalaganje nove verzije (po urejanju) pa storimo z metodo *put_object*. Tukaj je naprej potrebno sliko, ki je zapisana v začasni datoteki, odpreti.

Za komunikacijo z mobilno aplikacijo je bilo potrebno ustvariti dodatnega nadzornika, ki je precej podoben nadzorniku, ki upravlja z uporabniki. Potrebno je bilo izdelati akcije, ki bodo skrbele za vpis (avtentikacijo) in nalaganje posnete fotografije na uporabnikov račun. Za razliko od prej, pa bo v tem primeru komunikacija potekala s pomočjo JSON sporočil. Primer 4.15 prikazuje akcijo za vpis z mobilno aplikacijo.

Primer 4.15: Metoda za prijavo uporabnika

```
def create
   user = User.find_by(username: params[:session][:username])
   if user && user.authenticate(params[:session][:password])
      log_in user
      respond_to do |format|
         format.json { render :status => 200,
            :json => { :success => true,
            :info => 'Succesful login!',
            :data => { :user_data => current_user.id }
            }
      }
      end
   else
      respond_to do |format|
         format.json { render :status => 401,
            :json => { :success => false,
            :info => 'Login failed!',
            :data => {}
            }
      }
      end
   end
```

```
end
```

4.2.3 Pogled

Za prikaz vsebine (tvorbo HTML dokumenta) skrbi pogled prikazan v primeru 4.16. V splošnem lahko pogled vsebuje neke statične podatke, lahko pa se na podlagi podatkov posredovanih iz nadzornika sestavi ustrezen pogled in ga napolni z vsebino iz podatkovne baze. Osnovni metodi, ki se tukaj uporabljata sta *render* in *yield*. Prva metoda podpira modularno zgradbo HTML dokumenta, kar pomeni, da ni treba spisati celotnega HTML dokumenta. Tako lahko v posamenih datotekah zapišemo le posamezne dele in jih nato z ukazom *render* sestavimo v celoten pogled. Metoda *yield* pa je namenjena izpisu vsebine, ki je zapisana v metodi *provide*.

Primer 4.16: Osnova struktura naše aplikacije

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> I++ </title>
    <link
      href='https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300'
      rel='stylesheet' type='text/css'>
    <link
      href="https://fonts.googleapis.com/icon?family=Material+Icons"
      rel="stylesheet">
    <%= stylesheet_link_tag 'application', media: 'all',
      'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application',
      'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= render 'layouts/header' %>
```

```
      <div class="container">
        <%= yield %>
      </div>
      <%= render 'layouts/footer' %>
   </body>
</html>
```

V primeru 4.16 je prikazana osnovna struktura HTML dokumenta spletne aplikacije. Za prikaz vsebine poskrbi ukaz *<% yield %>* objet s posebnima značkama. Ti znački sta potrebni za uporabo Ruby programske kode v HTML dokumentu (.erb - embedded ruby). Metoda oziroma ukaz, ki je prav tako uporaben v HTML dokumentu je *link_to*, katerega uporaba je prikazana v primeru 4.17. Na podlagi njega se ustvari povezava (HTML element a), besedilo, ki postane povezava pa predstavlja prvi argument. V našem primeru je namesto besedila, kar ikona. Za njen prikaz pa je potreben dodaten ukaz (*raw*). Neka povezava lahko vsebuje kar opozorilo (angl. alert), pri čemer je potrebno nastaviti polje *data*. Povezavo lahko predstavlja tudi slika, pri čemer se za njen prikaz uporablja metoda *image_tag*, ki ustvari HTML element *img*.

Primer 4.17: Prikaz slike

```
<% if (current_user?(image.user) || !image.private?) ||
    current_user.admin? %>
   <li>
   <span>
   <% if current_user?(image.user) %>
      <span id="image_name" style="display: inline;"><%= image.name
         %></span>
      <%= link_to raw('<i class="material-icons">delete</i>'),
         image, method: :delete, data: { confirm: 'Are you sure
         you want to delete this image?' }, id: 'delete_image'%>
   <% end %>
   </span>
```

```
    <span>
      <%=link_to image_tag(image.source.to_s, :class =>
          'img-thumbnail'), {:controller => 'images', :action =>
          'show', :id => image.id} %>
    </span>
    </li>
<% end %>
```

Neposredno v .html.erb datoteki lahko uporabimo tudi ostale ukaze. V primeru 4.17 pred prikazom slike preverimo, za katerega uporabnika gre, torej če ima pravico do brisanja in ali je slika javna. Če je slika zasebna ali ne pripada trenutnemu uporabniku, ne bo prikazana. Razen, če je trenutni uporabnik administrator.

Ker razvijamo spletno aplikacijo, uporabljamo obrazce za vnos podatkov. Osnovni ukaz za tvorbo obrazcev je *form_for*. Poleg tega je potreben objekt, kamor se bodo zapisali podatki. V primeru 4.18 temu služi spremenljivka *@user*. Namenjena je polnjenju nekega objekta pri obrazcih ali še bolj pogosto hrambi podatkov, ki jih posreduje nadzornik pogledu za izpis. Z ukazi oblike *f.vnosno_polje* tvorimo ustrezna vnosna polja na podlagi katerih se kreira objekt. Simbol, ki se pojavi za tem ukazom, definira atribut, za čimer stoji tudi validacija vnosa. Vrstica *f.password_field* kreira HTML element za vnos gesla, ki pripada objektu *@user* (v nadzorniku je to razred *User*, torej model). Simbol *:password* pa določi, da gre za atribut *password* (geslo) v modelu. Gumb pa se kreira z ukazom *f.submit*. Programska koda je zapisana v primeru 4.18.

Primer 4.18: Forma za ponastavitev gesla

```
<% provide(:text, 'SETTINGS') %>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for @user do |f|%>
      <%= render 'shared/error_messages', object: f.object %>
```

```
        <%= f.label :password %>
        <%= f.password_field :password, class: 'form-control',
            placeholder: 'New password must contain 6 to 18
            characters, at least one digit' %>
        <br>
        <%= f.label :password_confirmation %>
        <%= f.password_field :password_confirmation, class:
            'form-control', placeholder: 'Match the previous input'
            %>
        <br>
        <%= f.submit 'RESET PASSWORD', class: 'btn btn-lg
            btn-primary' %>
      <% end %>
   </div>
</div>
```

4.3 Mobilna aplikacija

Poleg razvoja osnovne spletne aplikacije smo se odločili tudi za razvoj mobilne aplikacije, katere namen je podpora spletni aplikaciji. Uporabnik bi se preko mobilne aplikacije prijavil na svoj račun, nato pa bi imel možnost fotografiranja, nakar bi lahko naložil posneto fotografijo na svoj račun.

Po tehtnem premisleku smo se odločili, da bomo razvijali aplikacijo za operacijski sistem Android, predvsem zaradi prevladujočega tržnega deleža, poznavanja sistema in relativno enostavnega razvoja ter testiranja. Kot razvojno okolje smo uporabili Android Studio, ki temelji na programskem orodju JetBrains IntelliJ IDEA in programski jezik Java. Testiranja pa nismo opravili v emulatorju, ki ga podpira razvojno okolje, temveč na mobilnih napravah Samsung S6, z operacijskim sistemom Android OS Marshmallow in Lenovo A600 z operacijskim sistemom Android OS Lollipop.

4.3.1 Aktivnost LoginActivity

Razvoj mobilne aplikacije za operacijski sistem Android temelji na principu aktivnosti. Aktivnost je v bistvu prikazan vmesnik na zaslonu, ki ga vidi uporabnik. Neka aktivnost je v programski kodi zapisana kot razred, ki razširja razred *AppCompatActivity*. Osnovna aktivnost ima lahko več povratnih metod (angl. callback method), ki se izvedejo zaradi različnih dogodkov:

- *onCreate()* - se kliče ob kreiranju aktivnosti
- *onStart()* - ob vidnosti aktivnosti
- *onPause()* - se kliče ob premoru aktivnosti, nadaljuje se prejšnja
- *onStop()* - ob prenehanju vidnosti aktivnosti
- *onDestroy()* - preden se aktivnost uniči
- *onRestart()* - se kliče, ko se aktivnost znova zažene po prekinitvi

Prva oziroma osnovna aktivnost, ki se uporabniku prikaže ob zagonu aplikacije je aktivnost imenovana LoginActivity. Tukaj se mora za nadaljevanje uporabnik avtenticirati s svojim uporabniškim imenom in geslom. Kliče se metoda onCreate(), katere programska koda je prikazana v primeru 4.19.

Primer 4.19: Metoda onCreate()

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   this.requestWindowFeature(Window.FEATURE_NO_TITLE);
   this.getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
       WindowManager.LayoutParams.FLAG_FULLSCREEN);
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_login);
}
```

Prvi dve vrstici v metodi služita skrivanju zgornje orodne vrstice, namenjene prikazu ure, baterije in drugih obvestil v operacijskem sistemu Android.

Drugi dve pa zaženeta oziroma prikažeta grafični vmesnik aktivnosti zapisan v posebni datoteki activity_login.xml (xml datoteka z opisom vmesnika).

Ob pritisku na gumb za prijavo (LOGIN) se začne postopek avtentikacije. Najprej se pregleda, če sta vnosni polji zapolnjeni, drugače se to sporoči uporabniku z obvestilom, naj vpiše svoje podatke (Toast sporočilo). Nato se preveri, ali ima uporabnik vključeno internetno povezavo (primer 4.20), saj je zahtevana komunikacija s spletno aplikacijo, pri kateri se pošlje JSON sporočilo (primer 4.21), na podlagi katerega strežnik vrne bodisi sporočilo o uspešni prijavi (primer 4.22), bodisi sporočilo o neuspehu.

Primer 4.20: Metoda isOnline()

```
public boolean isOnline() {
   ConnectivityManager cm = (ConnectivityManager)
       getSystemService(Context.CONNECTIVITY_SERVICE);
   NetworkInfo netInfo = cm.getActiveNetworkInfo();
   return netInfo != null && netInfo.isConnectedOrConnecting();
}
```

Za kreiranje JSON sporočila oziroma objekta uporabljamo poseben razred, ki razširja razred *UrlJsonAsyncTask*. Metoda *doInBackgroud()* iz danih vpisnih podatkov sestavi pravilno obliko JSON sporočila.

Metoda *onPostExecute()* pa v primeru uspeha, ki ga vrne strežnik zopet v obliki JSON sporočila, zažene novo aktivnost imenovano WelcomeActivity, kjer je znan *id* vpisanega uporabnika, ki ima nato možnost nalaganja slike na svoj račun. V primeru neuspeha uporabnik do te aktivnosti ne dostopa.

Primer 4.21: JSON zahtevek

```
{"session": {{"username" : "rokzidarn"},
        {"password" : "geslo123"}}
}
```

Primer 4.22: JSON odgovor

```
{"success" : "true",
```

```
 "info" : "Succesful login!",
 "data" : {"user_data : "1"}
}
```

4.3.2 Aktivnost WelcomeAcitivity

V aktivnost za dobrodošlico, prikazano v primeru 4.23, pride uporabnik le ob uspešni prijavi. V prejšnji aktivnosti se ustvari objekt *Intent*, ki služi kot pasivna struktura za zagon nove aktivnosti, ki sproži prehod iz ene aktivnosti v drugo. K temu objektu se doda še en objekt imenovan *Bundle*, ki se uporablja za prenos podatkov med aktivnostmi. V ta objekt se zapiše *id* poslan s strani strežnika ob prijavi ter uporabniško ime, da v novi aktivnosti vemo, za katerega uporabnika gre.

Primer 4.23: Uspešna avtentikacija

```
String id = json.getJSONObject("data").getString("user_data");
Intent intent = new Intent(getApplicationContext(),
    WelcomeActivity.class);

Bundle b = new Bundle();
b.putString("user", mUserUsername);
b.putString("id", id);
intent.putExtras(b);

startActivity(intent);
finish();
```

V tej novi aktivnosti se pojavita dva gumba. Gumb CAMERA sproži Android aplikacijo za dostop do fotoaparata, s katero uporabnik posname fotografijo. Za to se uporabi nov objekt *Intent* vezan na zagon fotoaparata naprave. Gumb UPLOAD pa je namenjen prenosu posnete fotografije. Ta gumb prav tako kot gumb LOGIN preveri ustreznost polj (če je zapisano ime slike, izbrana dostopnost, in če je posneta fotografija), preden začne z

nalaganjem fotografije.

S pritiskom na gumb CAMERA lahko uporabnik posname fotografijo. Za tem se sproži metoda *onAcitivityResult()*, prikazana v primeru 4.24, ki v primeru uspeha na podlagi URI-ja (angl. Uniform Resource Identifier) pridobi sliko iz pomnilnika in jo pretvori v objekt *Bitmap*. Dodan je še poseben pogoj. V primeru, da je slika prevelika, torej je posneta z resolucijo večjo od 8 milijonov slikovnih enot (3264x2448), se izvede metoda, ki sliko zmanjša. To smo dodali zaradi predolgega nalaganja na strežnik in nepotrebne velike resolucije ter posledično prevelike zasedenosti prostora.

Preden se začne pretvorba v JSON sporočilo in nalaganje na strežnik, je treba sliko pretvoriti iz formata JPEG, ki se ne more pošiljati v JSON formatu v nizu znakov. Za to se uporablja metoda *Base64.encodeToString()*.

Primer 4.24: Priprava fotografije

```
super.onActivityResult(requestCode, resultCode, data);
Bitmap bitmapImage = (Bitmap) data.getExtras().get("data");

Uri imageUri = data.getData();
Bitmap bitmap = null;
try {
   bitmap =
       MediaStore.Images.Media.getBitmap(this.getContentResolver(),
       imageUri);
}
catch (Exception e) {
   e.printStackTrace();
   Log.e("URI to Bitmap problem!", "" + e);
}
imageView.setImageBitmap(bitmapImage);

if(bitmap.getWidth()>3264 || bitmap.getHeight()>3264){
   bitmap = getResizedBitmap(bitmap, bitmap.getWidth()/3,
       bitmap.getHeight()/3);
```

```
}

ByteArrayOutputStream baos = new ByteArrayOutputStream();
bitmap.compress(Bitmap.CompressFormat.JPEG, 100, baos);
byte[] imageBytes = baos.toByteArray();
mEncodedImage = Base64.encodeToString(imageBytes, Base64.DEFAULT);
Log.e("ENCODED IMAGE", mEncodedImage);
```

4.3.3 Android Manifest

Datoteka AndroidManifest.xml je namenjena definiranju vseh aktivnosti, ki jih ima na voljo aplikacija. Tukaj se določi, katera ikona bo vidna ob zagonu, katera bo glavna/začetna aktivnost, stil posamezne aktivnosti, orientacija in tako naprej. Naša aplikacija na primer v vseh aktivnostih uporablja orientacijo portret (*screenOrientation: portrait*), kar onemogoča zasuk zaslona. Dodatno pa je potrebno tudi definirati dovoljenja, ki jih potrebuje aplikacija za delovanje, kot so na primer dostop do interneta, lokacije in kontaktov.

4.3.4 Grafični vmesnik

Grafični vmesnik aplikacije smo oblikovali s pomočjo XML datotek za posamezno aktivnost. Na podlagi Android imenskega prostora smo definirali vnosna polja, gumbe in ikone s pripadajočimi atributi s katerimi smo še natančneje oblikovali izgled. Primer 4.25 prikazuje ustvarjanje gumba.

Primer 4.25: Android gumb

```
<Button
   android:id="@+id/btnLogin"
   android:onClick="login"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:layout_marginTop="20dip"
   android:background="@color/turqoise"
```

```
android:text="@string/btn_login"
android:textColor="@color/white" />
```

Poglavje 5

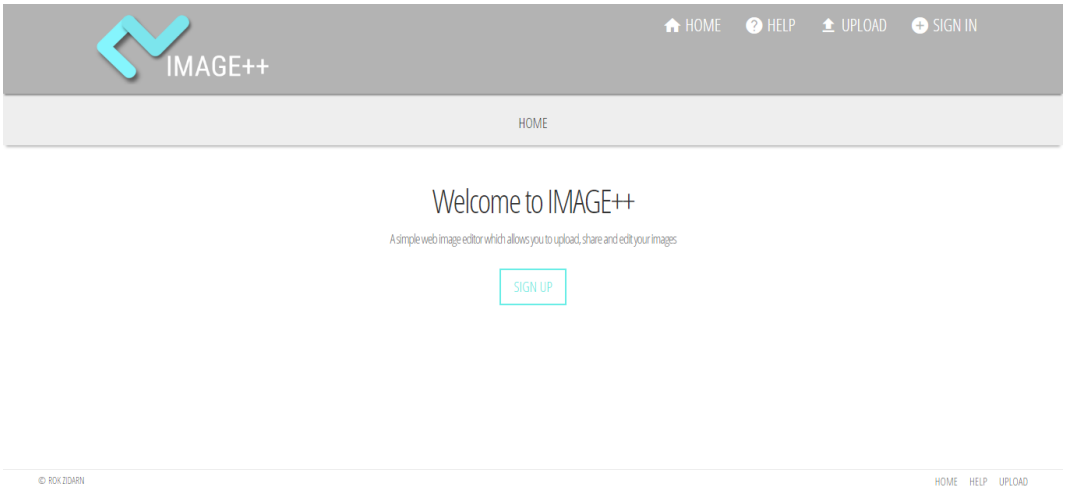
Delovanje aplikacije

V tem poglavju bomo predstavili spletno in mobilno aplikacijo z vidika uporabnikov. Torej od prijave in nalaganja, do uporabe glavne funkcionalnosti, urejanja slik. Aplikacijo lahko uporabljajo vsi z dostopom do spleta. Dostopna je na naslovu *https://imageplusplus.herokuapp.com/*, kjer se uporabnikom najprej prikaže glavna stran. Tukaj je pozdravno sporočilo, opis, ki pojasnjuje kakšna aplikacija sploh to je in gumb, ki jih preusmeri na stran, kjer se lahko registrirajo.

5.1 Glavna stran aplikacije

Ob dostopu do spletne aplikacije se najprej prikaže glavna stran. Vsaka stran je sicer sestavljena iz glave, vsebine in noge, kot je prikazano na sliki 5.1. Glava in noga sta skozi aplikacijo enaki. Sicer se glava pri registriranih in neregistriranih uporabnikih razlikuje. Glava je sestavljena iz treh komponent. Na levi strani se nahajata logotip in ime aplikacije. Malce nižje in na sredini se izpisuje naslov podstrani, na kateri se uporabnik trenutno nahaja. Glavna stran predstavlja tudi domačo stran vseh uporabnikov, zato je izpisan naslov *HOME*. Navigacijski meni se nahaja v zgornjem desnem kotu, kjer so izrisane ikone, kjer vsaka predstavlja preusmeritev na ustrezno podstran. Na sliki 5.1 prvi gumb, v obliki ikone, predstavlja glavno/domačo stran (*HOME*), drugi

je *HELP*, kjer so natančnejše informacije v zvezi z uporabo aplikacije. Naslednji gumb preusmeri uporabnika na podstran, kjer poteka lokalno urejanje slik. Na podstrani *UPLOAD* lahko vsak ureja naloženo fotografijo, ki jo mora po urejanju znova naložiti na svoj računalnik, saj se nikamor ne shrani. Omogočeni so le nekateri slikovni učinki, v primeru dostopa do ostalih slikovnih učinkov, ki jih ponuja aplikacija, pa se mora uporabnik prijaviti. To stori s klikom na zadnji gumb (*SIGN IN*). Seveda mora biti predhodno registiran, to pa stori s klikom na osrednji gumb viden na glavni/domači strani, kjer piše *SIGN UP*.

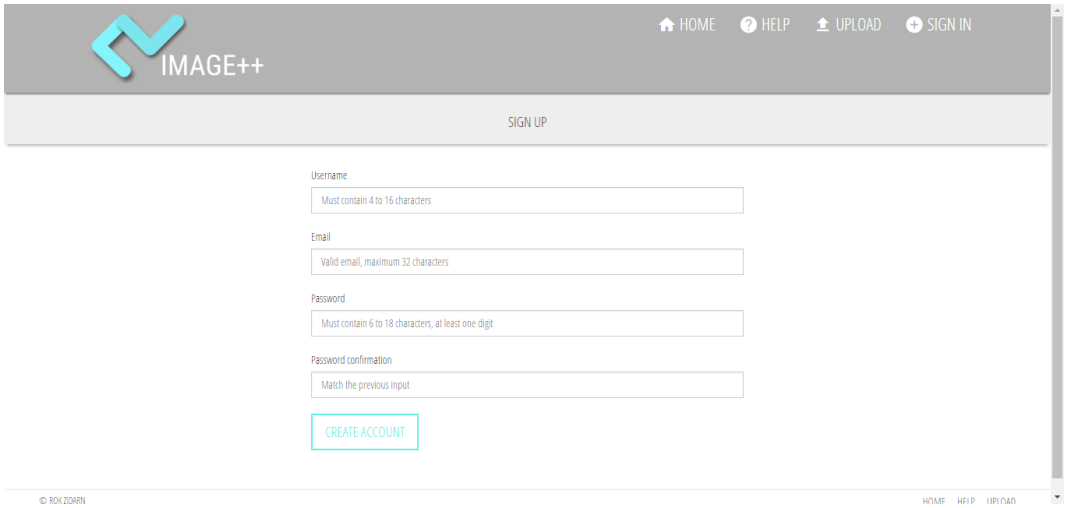


Slika 5.1: Glavna/začetna stran

5.2 Registracija in prijava

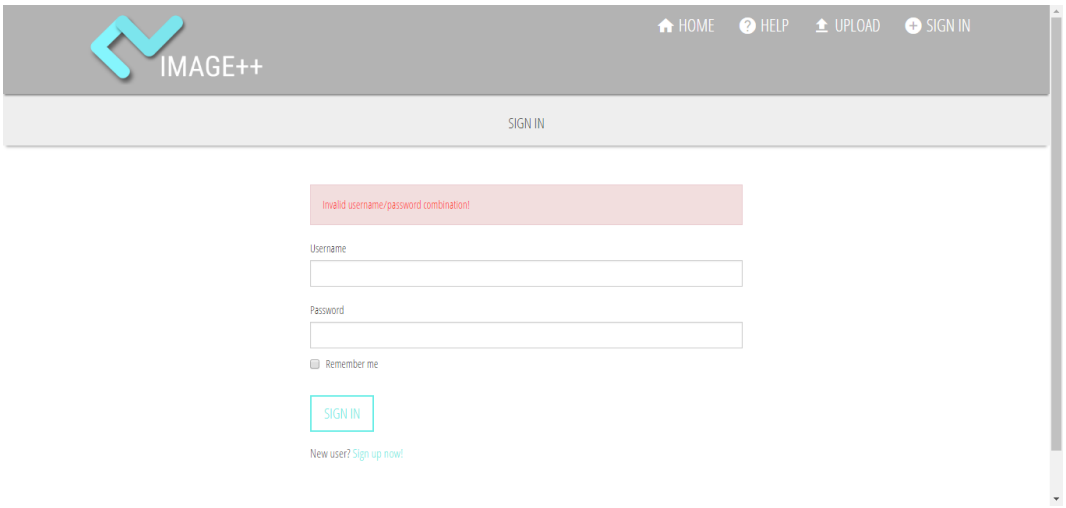
Do registracije uporabnik dostopa s klikom na gumb *SIGN UP*. Stran je prikazana na sliki 5.2. Tukaj mora izpolniti štiri polja. Najprej mora ustvariti uporabniško ime, ki v aplikaciji trenutno še ne obstaja, saj na podlagi njega poteka prijava. Vsebuje lahko vse črke v angleški abecedi (male ali velike tiskane), vsa števila, znak _ (angl. underscore) in piko. Nato mora v naslednje polje zapisati veljaven elektronski naslov. Polji, ki sledita sta potrebni

za geslo. V prvo se vpiše geslo, ki mora vsebovati med šest in osemnajst alfanumeričnih znakov. Med vnešenimi znaki mora biti vsaj ena številka. V drugo pa mora uporabnik zapisati enako geslo kot v prvo, saj je namenjeno potrditvi gesla.



Slika 5.2: Registracija

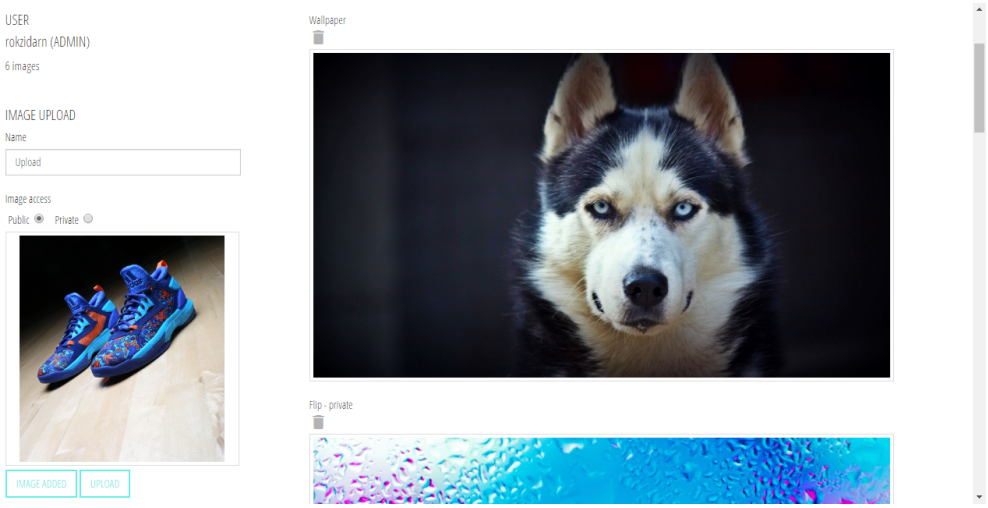
Po uspešni registraciji lahko začne posameznik uporabljati aplikacijo in nalagati slike. Pred tem se mora prijaviti na obstoječ račun z uporabniškim imenom in geslom. Pri tem ima tudi možnost funkcije "remember me", s čimer se ustvari piškotek, ki omogoča avtomatsko prijavo ob naslednjem dostopu. Če se slučajno zmoti pri vpisovanju kombinacije uporabniškega imena in gesla, avtentikacija ne uspe, tako da se izpiše ustrezno sporočilo, kot je prikazano na sliki 5.3. Podobno je pri ostalih vnosih s strani uporabnika, bodisi pri registraciji ali nalaganju slike.



Slika 5.3: Neuspešna prijava

5.3 Nalaganje slik

Nalaganje slik poteka na domači strani registriranega uporabnika, ki je prikazana na sliki 5.4. Na desni strani ima prikazane slike, ki jih je že naložil, forma za nalaganje nove slike pa se nahaja na levi strani. Pri nalaganju nove slike mora uporabnik vpisati ime slike, ki ne sme imeti več kot 24 znakov. Odločiti se mora tudi za dostopnost slike, torej ali bo na voljo vsem v galeriji (javna dostopnost), ali bo vidna le njemu (zasebna dostopnost). Izbiro slike doseže s klikom na gumb *SELECT IMAGE*. S tem se odpre knjižnica s slikami na njegovem računalniku, kjer nato s klikom na želeno sliko in potrditivijo naloži predogled v aplikacijo. Če se je zmotil, lahko ponovno klikne gumb, kjer sedaj piše *IMAGE ADDED* in znova izbere sliko. S klikom na gumb *UPLOAD* pa se začne izbrana slika nalagati v aplikacijo, kjer bo sedaj trajno hranjena in namenjena urejanju. Posamezno sliko lahko uporabnik kasneje tudi izbriše s klikom na gumb, ki ga predstavlja ikona smetnjaka nad sliko. V primeru kakršne koli napake, bodisi neustrezno ime slike, ali morda celo napačen slikovni format (ustrezni so le JPG oziroma JPEG, GIF in PNG) se slika ne naloži, uporabniku pa se prikaže opozorilo o napaki.

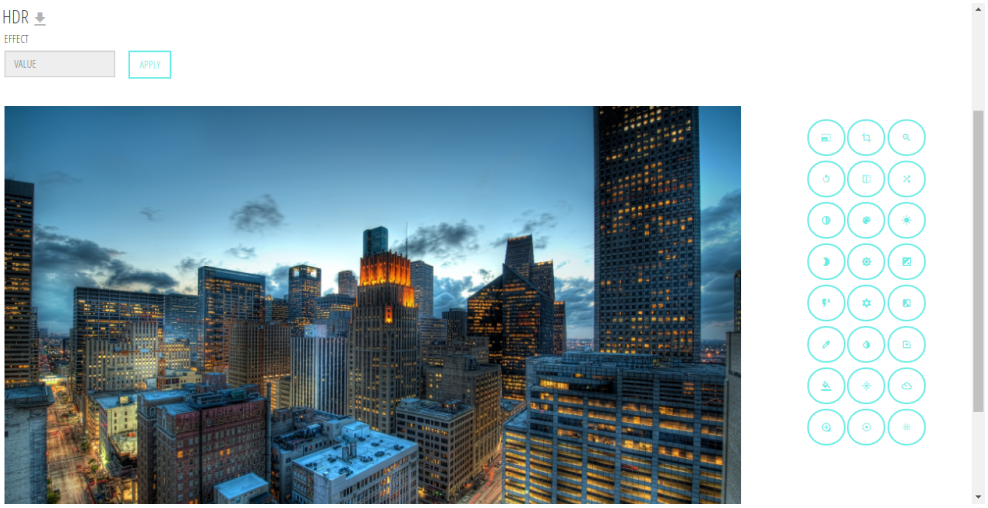


Slika 5.4: Domača stran registriranega uporabnika

5.4 Urejanje slik

Do strani za urejanje neke slike, ki je prikazana na sliki 5.5, pridemo tako, da kliknemo na želeno sliko na strani uporabnika. Če je trenutni uporabnik, ki je vpisan, lastnik slike, jo lahko ureja. Drugače si jo lahko zgolj ogleda, oziroma naloži na računalnik. Ob desni strani se nahaja meni z orodji, ki so namenjeni urejanju. Uporabnik lahko izbira med štiriindvajsetimi različnimi slikovnimi učinki. Na voljo ima razne filtre, manipulacijo z barvami, svetlobo in ostalimi lastnostmi slike. Filtri se uporabijo tako, da se le klikne na gumb, nato pa se spremembo še potrdi z dodatnim klikom na gumb nad sliko. Razna manipulacija z barvami in podobno temelji na drsnikih, tako da uporabniku ni treba vpisovati vrednosti. Za nekaj slikovnih učinkov pa je potrebno vnesti oziroma s klikom izbrati številčno vrednost, kot je na primer pri spremembi velikosti. Smiselna vrednost za to polje predstavljajo števila med ena in sto, kjer se velikost slike spreminja glede na odstotek trenutne velikosti.

Možnosti urejanja lastnosti slik so naslednje: sprememba velikosti (angl. resize), rezanje (angl. crop), povečava z lupo (angl. zoom), obračanje (angl. rotate), obračanje po horizontali (angl. flip), zrcaljenje (angl. mirror), gla-



Slika 5.5: Urejanje slike

jenje (angl. blur), ostrenje (angl. sharpen) in implozija (angl. implode).

Možne manipulacije z barvo so: sprememba odtenka barve (angl. hue), sprememba nasičenosti (angl. saturation), sprememba intezitete svetlosti (angl. lightness), sprememba svetlosti (angl. brightness), sprememba kontrasta (angl. contrast) in sprememba game (angl. gamma).

Aplikacija ima na voljo tudi nekaj filtrov: avto (angl. autocorrect), sepija (angl. sepia), črno-belo (angl. black and white), barvanje (angl. colorize), negacija (angl. invert), olje (angl. oil), lomo (angl. lomo), toaster (angl. toaster) in gotham (angl. gotham).

Nekateri izmed možnih slikovnih učinkov/efektov so prikazani na slikah 5.6 in 5.7. Na sliki 5.6 so prikazani: obračanje po horizontali (zgoraj levo), sprememba odtenka barve (zgoraj desno), sprememba kontrasta (spodaj levo) in sprememba svetlosti (spodaj desno). Na sliki 5.7 pa so prikazani: sepija filter (zgoraj levo), lomo filter (zgoraj desno), olje (spodaj levo) in glajenje (spodaj desno).

Slika 5.6: Rezultat urejanja slik z uporabo obračanja po horizontali (zgoraj levo), spremembe odtenka barve (zgoraj desno), spremembe kontrasta (spodaj levo) in spremembe svetlosti (spodaj desno).

Slika 5.7: Rezultat urejanja slik z uporabo sepija filtra (zgoraj levo), lomo filtra (zgoraj desno), efekta olja (spodaj levo) in glajenja (spodaj desno).

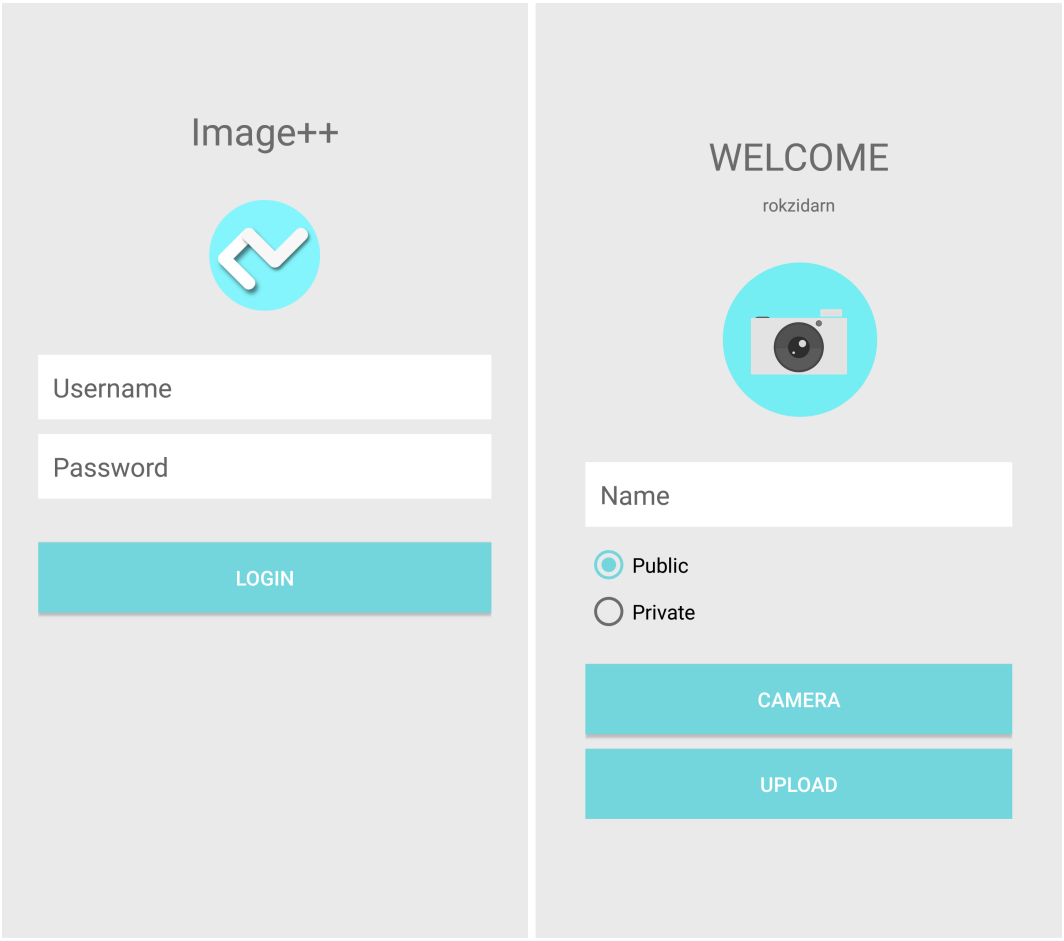
5.5 Administrator

Posebno vlogo v aplikaciji ima uporabnik, ki je administrator. Ta uporabnik lahko pregleduje vse slike, tudi tiste, ki so zasebne, in v primeru zaznanih neskladnosti oziroma neprimernih fotografij, lahko izbriše uporabnika. To stori s klikom na ikono smetnjaka, ki je v bistvu gumb za brisanje uporabnika. Do seznama vseh uporabnikov dostopa s klikom na gumb z ikono uporabnikov (*USERS*).

5.6 Mobilna aplikacija

Mobilno aplikacijo si lahko uporabnik naloži s spletne aplikacije na podstrani *HELP*). Namenjena je registriranim uporabnikom in nalaganju fotografij neposredno z mobilne naprave. Za delovanje je potrebna internetna povezava in naprava, na kateri teče Android operacijski sistem.

V aplikacijo se mora uporabnik najprej prijaviti s svojim uporabniškim imenom in geslom. Temu služi prva aktivnost vidna na sliki 5.8 (levo). Če avtentikacija uspe, se odpre naslednja aktivnost (slika 5.8 (desno)) namenjena nalaganju slike na uporabniški račun. S pritiskom na gumb *CAMERA* se aktivira kamera mobilne naprave, s katero lahko uporabnik posname fotografijo. Nato mora izpolniti polje namenjeno imenu fotografije in izbiri dostopnosti slike (javna ali zasebna). Vsa polja so obvezna, tudi pri prijavi. V primeru, ko pride do napak se izpiše sporočilo. S pritiskom na gumb *UPLOAD* se začne prenos slike na strežnik in shranjevanje na uporabniški račun.



Slika 5.8: Aktivnost za prijavo (levo) in aktivnost za nalaganje slike (desno)

Poglavje 6

Zaključek in nadaljne delo

V diplomskem delu je prikazan razvoj spletne aplikacije v ogrodju ROR. Kot dodaten del, ki služi podpori spletni aplikaciji, pa je razvita tudi mobilna aplikacija za sistem Android OS. Najprej so predstavljene tehnologije, ki smo jih uporabili pri razvoju. Opisane so glavne značilnosti in osnove, ki so prikazane na primerih. Nato je opisan načrt podatkovnega modela, ki služi hrambi podatkov. Dejanski razvoj pa je natančneje opisan v naslednjem poglavju, kjer lahko bralec vidi aplikacijo s strani programske kode. Če želi spoznati delovanje spletne aplikacije oziroma uporabo, je ta opisana v petem poglavju. Na voljo so tudi slike, ki prikazujejo grafični vmesnik.

Cilj diplomske naloge je bil razviti spletno aplikacijo za urejanje slik. Aplikacija je uspešno izdelana in objavljena na spletu. Delujoč je osnovni del urejanja slik namenjen vsem uporabnikom aplikacije, možna je tudi registracija in trajno hranjenje slik. Poleg spletne aplikacije pa je bila razvita še mobilna aplikacija, ki prav tako deluje uspešno. Možna je prijava z obstoječim uporabniškim imenom in geslom.

Na voljo je štiriindvajset različnih načinov urejanja od spreminjanja lastnosti slik, kot je velikost, rotacija, orientacija in podobno, do manipulacije z barvami. Možna je tudi uporaba raznih filtrov.

Glavne prednosti razvite aplikacije pred drugimi tovrstnimi aplikacijami so, da je aplikacija prostodostopna, da omogoča aplikacija tako hranjenje

slik na oddaljenem strežniku kot tudi urejanje shranjenih slik. Poleg tega pa je bila razvita tudi aplikacija za mobilne naprave, ki omogoča zajem slik in prenašanje slik na oddaljen strežnik.

Pri koncu razvoja smo začeli razmišljati tudi o raznih izboljšavah. Prva izmed teh bi bila hitrejše urejanje slik na spletu, ki deluje, vendar sama funkcionalnost ni ravno najhitrejša, kljub temu pa uporabniku omogoča enostavno urejanje shranjenih slik. Hitrost delovanja je predvsem odvisna od hitrosti nalaganja (angl. upload speed) slik na splet. Proces urejanja je namreč sestavljen iz več korakov. Najprej se slika pošlje v POST zahtevku. Slika se nato odpre in uredi, nakar je potrebno zapisovanje na disk. Problematičen je del, ki z AWS storitve najprej izbriše prejšnjo verzijo in ponovno naloži novo verzijo slike. Dobrodošlo bi bilo razviti kak Ruby gem, ki bi pohitril posodabljanje datotek na oblačnih storitvah. Glede na to, da ima podjetje Apple z mobilnimi napravami iPhone tudi velik tržni delež, bi bilo zaželeno razviti mobilno aplikacijo tudi za platformo iOS. V določenih primerih pride prav tudi nalaganje in urejanje večjega števila slik hkrati (angl. bulk editing), ki bi ga bilo tudi dobro implementirati. Možno bi bilo urediti tudi deljenje slik le določenim uporabnikom. V prihodnosti bi bilo smiselno tudi razviti podporo za več jezikov uporabniškega vmesnika, saj trenutna verzija namreč uporablja angleščino. Seveda pa bi bilo smiselno dodati čim večje število slikovnih učinkov.

Literatura

[1] Adobe photoshop cc. http://www.adobe.com/si/products/photoshop.html. Accessed: 2016-08-11.

[2] Amazon s3. http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html. Accessed: 2016-08-11.

[3] Amazon web services. https://aws.amazon.com/. Accessed: 2016-08-11.

[4] Android studio ide. https://developer.android.com/studio/index.html. Accessed: 2016-08-11.

[5] Aws sdk. http://docs.aws.amazon.com/sdkforruby/api/index.html. Accessed: 2016-08-19.

[6] Bootstrap framework. http://getbootstrap.com/. Accessed: 2016-08-11.

[7] Data types. https://en.wikipedia.org/wiki/Data_type. Accessed: 2016-08-11.

[8] Flickr. https://www.flickr.com/. Accessed: 2016-08-11.

[9] Gitbash. https://git-for-windows.github.io/. Accessed: 2016-08-11.

[10] glfx library. http://evanw.github.io/glfx.js/. Accessed: 2016-08-11.

[11] Google material icons. https://design.google.com/icons/. Accessed: 2016-08-11.

[12] Heroku cloud application platfrom. https://www.heroku.com/. Accessed: 2016-08-11.

[13] Imgur. http://imgur.com/. Accessed: 2016-08-11.

[14] Intellij idea ide. https://www.jetbrains.com/idea/. Accessed: 2016-08-11.

[15] Ipiccy photo editor. http://ipiccy.com/. Accessed: 2016-08-11.

[16] Jasny bootstrap. http://www.jasny.net/bootstrap/javascript/#fileinput. Accessed: 2016-08-11.

[17] jquery library. https://jquery.com/. Accessed: 2016-08-11.

[18] Minimagick ruby gem. https://github.com/minimagick/minimagick. Accessed: 2016-08-11.

[19] Pixlr editor. https://pixlr.com/. Accessed: 2016-08-11.

[20] Ruby. https://www.ruby-lang.org/en/. Accessed: 2016-08-11.

[21] Ruby on rails. http://rubyonrails.org/. Accessed: 2016-08-11.

[22] Usage of server-side programming languages. https://w3techs.com/technologies/overview/programming_language/all. Accessed: 2016-08-11.

[23] David Geer. Will software developers ride ruby on rails to success? *Computer*, 39(2):18–20, 2006.

[24] Michael Hartl. *Ruby on rails tutorial: learn Web development with rails.* Addison-Wesley Professional, 2015.

[25] LLC ImageMagick Studio. Imagemagick, 2008.