UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matjaž Kraševec

# RAZVOJ SPLETNE APLIKACIJE ZA ANALIZO UPORABNIŠKIH PROFILOV NA FACEBOOKU

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2013

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matjaž Kraševec

# RAZVOJ SPLETNE APLIKACIJE ZA ANALIZO UPORABNIŠKIH PROFILOV NA FACEBOOKU

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Luka Šajn

Ljubljana, 2013

*Besedilo je oblikovano z urejevalnikom besedil LaTeX.*

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

*Tržaška 25*
*1001 Ljubljana, Slovenija*
*telefon: 01 476 84 11*
*01 476 83 87*
*faks: 01 426 46 47*
*01 476 87 11*
*www.fri.uni-lj.si*
*e-mail: dekanat@fri.uni-lj.si*

Št. naloge: 00400/2013
Datum: 03.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MATJAŽ KRAŠEVEC**

Naslov: **RAZVOJ SPLETNE APLIKACIJE ZA ANALIZO UPORABNIŠKIH PROFILOV NA FACEBOOKU**

**DEVELOPMENT OF A WEB APPLICATION FOR THE ANALYSIS OF FACEBOOK USER PROFILES**

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

V diplomskem delu poiščite način programskega dostopa do podatkov Facebookovega socialnega grafa in razvijte aplikacijo za analizo uporabniških profilov. Izdelajte uporabniški vmesnik, preko katerega se lahko uporabnik vpiše v aplikacijo s svojim uporabniskim računom na Facebooku. Uporabniku predstavite analizo njegovega profila, ki naj vsebuje informacije, neočitne iz vsakdanje uporabe Facebooka. Poiščite primeren način prikaza teh informacij in aplikaciji pri razvoju določite ustrezno arhitekturo.

Mentor:

doc. dr. Luka Šajn

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:

prof. dr. Nikolaj Zimic

# Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matjaž Kraševec,

z vpisno številko 63060229,

sem avtor diplomskega dela z naslovom:

*Razvoj spletne aplikacije za analizo uporabniških profilov na Facebooku*

*Development of a web application for the analysis of Facebook user profiles*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Luka Šajna,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 11. oktobra 2013 Podpis avtorja:

*Zahvaljujem se mentorju, družini in vsem ostalim, ki so me podpirali med študijem in mi kakorkoli pomagali pri diplomskem delu.*

# Kazalo

# Seznam kratic

- .NET – Microsoftovo ogrodje za razvoj programske opreme (angl. *.NET Framework*). Več informacij na naslovu: http://www.microsoft.com/net
- ADO.NET – komponenta .NET-a z naborom storitev za dostop do podatkov. Več informacij na naslovu: http://msdn.microsoft.com/en-us/library/aa286484.aspx
- AJAX – nabor tehnik za razvoj asinhronih spletnih aplikacij (angl. *asynchronous JavaScript and XML*). Več informacij na naslovu: http://www.w3schools.com/ajax
- API – programski vmesnik (angl. *application programming interface*). Več informacij na naslovu: http://en.wikipedia.org/wiki/Application_programming_interface
- ASP.NET – ogrodje za razvoj spletnih aplikacij. Več informacij na naslovu: http://www.asp.net
- C# – preprost, sodoben, splošnonamenski in objektno orientiran programski jezik za razvoj aplikacij na ogrodju .NET. Več informacij na naslovu: http://msdn.microsoft.com/en-us/library/vstudio/kx37x362.aspx
- CSRF – ponarejanje spletnih zahtev (angl. *cross-site request forgery*). Več informacij na naslovu: http://en.wikipedia.org/wiki/Cross-site_request_forgery
- CSS – jezik za izdelavo prekrivnih slogov, v katerih je zapisana oblika spletne strani (angl. *Cascading Style Sheets*). Več informacij na naslovu: http://www.w3schools.com/css
- D3 – knjižnica JS za obdelavo podatkovnih dokumentov (angl. *Data-Driven Documents*). Več informacij na naslovu: http://d3js.org

- DAO – objekt oz. vmesnik za manipulacijo in dostop do podatkovne baze (angl. *data access object*). Več informacij na naslovu:
  http://en.wikipedia.org/wiki/Data_access_object
- DLR – izvajalno okolje z naborom storitev za dinamične jezike (angl. *dynamic language runtime*). Več informacij na naslovu:
  http://msdn.microsoft.com/en-us/library/dd233052.aspx
- DO – dinamični objekt. Lastna kratica.
- DRY – princip programiranja s čim manjšim ponavljanjem kode (angl. *don't repeat yourself*). Več informacij na naslovu:
  http://en.wikipedia.org/wiki/Don't_repeat_yourself
- GPS – sistem za pozicioniranje (angl. *Global Positioning System*). Več informacij na naslovu: http://en.wikipedia.org/wiki/Global_Positioning_System
- HTML – označevalni jezik na osnovi XML, namenjen oblikovanju večpredstavnostnih dokumentov za prikaz v spletnem brskalniku (angl. *HyperText Markup Language*). Več informacij na naslovu: http://www.w3schools.com/html
- IIS – Microsoftov spletni strežnik (angl. *Internet Information Services*). Več informacij na naslovu: http://www.iis.net
- ISO – Mednarodna organizacija za standardizacijo (angl. *International Organization for Standardization*). Več informacij na naslovu: http://www.iso.org
- JS – skriptni jezik za ustvarjanje interaktivne ali animirane vsebine spletnih strani (angl. *JavaScript*). Več informacij na naslovu: http://www.w3schools.com/js
- JSON – tekstovni standard za izmenjavo podatkov (angl. *JavaScript Object Notation*). Več informacij na naslovu: http://www.json.org
- LINQ-to-SQL – komponenta .NET-a, ki zagotavlja izvajalno infrastrukturo za upravljanje z relacijskimi podatki (angl. *Language-Integrated Query for Relational Data*). Več informacij na naslovu:
  http://msdn.microsoft.com/en-us/library/bb386976.aspx
- OKG – spletna aplikacija, ki smo jo razvili v diplomskem delu (*ObrazoKnjigoGled*). Lastna kratica.

- psdmmm. – povprečje, standardna deviacija, minimum, maksimum in mediana. Lastna kratica.
- REST – arhitekturni slog za porazdeljene hipermedijske sisteme (angl. *representational state transfer*). Več informacij na naslovu: http://en.wikipedia.org/wiki/Representational_state_transfer
- SDK – komplet za razvoj programske opreme (angl. *software development kit*). Več informacij na naslovu: http://en.wikipedia.org/wiki/Software_development_kit
- SPI – spletni uporabniški vmesnik, ki se prilega na eno samo spletno stran (angl. *single-page interface*). Več informacij na naslovu: http://en.wikipedia.org/wiki/Single-page_application
- SQL – strukturiran povpraševalni jezik za delo s podatkovnimi bazami (angl. *Structured Query Language*). Več informacij na naslovu: http://www.w3schools.com/sql
- SVG – vektorski slikovni format za dvodimenzionalno grafiko na osnovi XML s podporo za interakcijo in animacijo (angl. *Scalable Vector Graphics*). Več informacij na naslovu: http://www.w3schools.com/svg
- URL – enolični krajevnik vira oz. spletni naslov (angl. *uniform resource locator*). Več informacij na naslovu: http://en.wikipedia.org/wiki/Uniform_resource_locator
- UTC – univerzalni koordinirani čas (angl. *Coordinated Universal Time*). Več informacij na naslovu: http://en.wikipedia.org/wiki/Coordinated_Universal_Time
- WCF – komponenta .NET-a za razvoj storitveno usmerjenih aplikacij (angl. *Windows Communication Foundation*). Več informacij na naslovu: http://msdn.microsoft.com/en-us/library/dd456779.aspx
- WPF – Microsoftovo ogrodje za izdelavo grafičnih uporabniških vmesnikov (angl. *Windows Presentation Foundation*). Več informacij na naslovu: http://msdn.microsoft.com/en-us/library/ms754130.aspx
- XML – razširljivi označevalni jezik (angl. *Extensible Markup Language*). Več informacij na naslovu: http://www.w3schools.com/xml

# Povzetek

V diplomski nalogi so opisani razvoj, delovanje in uporaba aplikacije za analizo uporabniških profilov na Facebooku. Prikazana je razdelitev večjega razvojnega problema na manjše dele in opisana izbrana aplikacijska arhitektura. Predstavljena je implementacija asinhronega spletnega uporabniškega vmesnika in objektov za generacijo dinamične vsebine. Preučeni so struktura Facebookovega socialnega grafa ter postopek avtentikacije uporabnikov in način programskega dostopa do Facebookovih podatkov z uporabo Graph API-ja. Opisana je izdelava podatkovne baze za hranitev uporabniških podatkov in podrobno obravnavan razvit algoritem za prenos uporabniških podatkov s Facebooka vanjo. Predstavljena sta vsebina analitičnega poročila, ki si ga uporabniki lahko zgenerirajo v razviti aplikaciji, in implementiran postopek izrisa vizualizacij z uporabo API-ja Google Charts in knjižnice Data-Driven Documents. Opisana je razvita knjižnica, namenjena obratnem geokodiranju in drugim izračunom, povezanim z geolokacijami. Diplomska naloga se zaključi z navodili za uporabo aplikacije, rezultati meritev časovne zahtevnosti algoritma za prenos podatkov in povzetkom opravljenega dela z nekaj idejami za izboljšave in nadaljnje delo.

## Ključne besede:

razvoj spletne aplikacije, Facebook, Graph API, večslojna arhitektura, spletni uporabniški vmesnik, vmesnik SPI, avtentikacija uporabnikov, prenos podatkov, dinamično programiranje, analiza podatkov, vizualizacija, geokodiranje.

# Abstract

The thesis demonstrates the development, operation, and usage of an application for the analysis of Facebook user profiles. A mayor development problem is broken down into smaller parts and the selected application architecture is described. The implementation process of an asynchronous web user interface, and of objects used for dynamic content generation is presented. The structure of the Facebook's social graph is studied, and a user authentication process and a method of accessing Facebook's data via the Graph API are described. The development of a database for the storage of user data is illustrated, and an algorithm for the transfer of user data from Facebook to the database is described in detail. The contents of the analytical report which users can generate in the developed application, and our method of visualization generation via the Google Charts API and the Data-Driven Documents library are presented. A library developed for the purpose of reverse geocoding and other calculations related to geolocations is described. In the last part of the thesis, application usage instructions, measurement results of the time complexity of the data transfer algorithm, and a summary of the work done with a few ideas for improvements and future work are provided.

## Ključne besede:

web application development, Facebook, Graph API, multi-tier architecture, web user interface, single-page interface, user authentication, data transfer, dynamic programming, data analysis, visualization, geocoding.

# Poglavje 1

# Uvod

Družabno spletno stran *Facebook*[1], ustanovljeno leta 2004 [1], danes aktivno uporablja že približno milijarda ljudi [2]. Uporabniki si na spletni strani ustvarijo profil, na katerem lahko objavijo osebne podatke, sporočila, slike, itd., ter preko njega dodajo druge uporabnike kot prijatelje [3].

V diplomski nalogi nas je zanimalo, kako je razvijalcem omogočen programski dostop do podatkov na Facebooku, ter ali lahko iz teh podatkov pridobimo informacije, ki niso očitne iz vsakodnevne uporabe Facebooka. Odločili smo se za razvoj aplikacije, ki ob želji uporabnika dostopi do podatkov njegovega profila na Facebooku, jih prenese in shrani v lokalno podatkovno bazo, ustrezno obdela in uporabniku predstavi v analitičnem poročilu. Razvito aplikacijo smo poimenovali *ObrazoKnjigoGled* oz. krajše *OKG*.

Ob pregledu obstoječih aplikacij s podobno funkcionalnostjo smo zasledili predvsem poslovne rešitve, namenjene analitiki profilov *Facebook Page*, tj. javnim profilom, namenjenim komercialni rabi, poslovnim uporabnikom, slavnim osebam, organizacijam, produktom, podjetjem, ipd.[2] Preprosto analitiko profilov tega tipa ponuja Facebook sam pod imenom *Page Insights*[2], ko govorimo o profilih zasebnih uporabnikov, t.i. profilih *Personal timeline*[2], pa te funkcionalnosti na Facebooku ni na voljo. Iz tega razloga smo se odločili v diplomski nalogi dati poudarek podatkom zasebnih uporabniških profilov. Med obstoječimi rešitvami, ki se osredotočajo nanje, smo zasledili le aplikacijo *Wolfram Alpha* podjetja Wolfram Research, ki od avgusta 2012 pod imenom *Personal Analytics for Facebook*[3] vključuje možnost analize zasebnih profilov na Facebooku [4]. Omenjena aplikacija nam je pri razvoju aplikacijske komponente za analizo in prikaz podatkov (Poglavje 3.6)

[1] Dostopno na naslovu: https://www.facebook.com
[2] Več informacij na naslovu: https://www.facebook.com/help
[3] Dostopno na naslovu: http://www.wolframalpha.com/facebook

predstavljala temeljno referenco.

V Poglavju 2 smo opisali tehnologije in orodja, ki smo jih uporabljali pri delu, ter v njegovih podpoglavjih opisali delo s programskim okoljem Facebook Platform. V Poglavju 3 smo nadaljevali z nekoliko širšim opisom razvoja, arhitekture, komponent in delovanja aplikacije *OKG*, čemur v Poglavju 4 sledijo preprosta navodila za uporabo in opis meritev časovne zahtevnosti razvitega Algoritma za prenos podatkov z rezultati. Na koncu, v Poglavju 5, smo povzeli in komentirali svoje delo ter predstavili nekaj idej za izboljšave aplikacije oz. nadaljnji razvoj.

# Poglavje 2

# Tehnologije in orodja

Aplikacijo *ObrazoKnjigoGled* smo razvili v razvojnem okolju *Microsoft Visual Studio Ultimate 2010* na prenosniku *HP Compaq nc8430* s konfiguracijo, razvidno iz Tabele 2.1.

Pri implementaciji zaledne procesne logike aplikacije *OKG* smo uporabili programski jezik C# z ogrodjem .NET 4.0. Pri izbiri verzije ogrodja nam je pomemben dejavnik predstavljala podpora DLR-ju oz. dinamičnim tipom `dynamic` [5, 6].

Za shranjevanje podatkov smo uporabili sistem za upravljanje podatkovnih baz *Microsoft SQL Server 2008* z orodjem za konfiguracijo in administracijo *Microsoft SQL Server Management Studio 2008*. Lokalne podatkovne baze smo z zaledno aplikacijsko logiko povezali z uporabo LINQ-to-SQL-a, ki temelji na tehnologiji ADO.NET in objektih `DataContext` [7, 8]. Z uporabo LINQ-to-SQL-a smo se izognili pisanju eksplicitnih poizvedb SQL v kodi in uporabi klasičnih, za implementacijo nekoliko nerodnejših, objektov ADO.NET-a za branje in modifikacijo podatkov, kot so `DataSet`, `DataReader` in `DataAdapter` [9].

Pri razvoju obličja aplikacije *OKG*, tj. uporabniškega vmesnika, smo nad ogrodjem

| komponenta | konfiguracija |
|---|---|
| procesor | Intel Mobile Core 2 Duo T7200 ($2 \times 2.00GHz$) |
| pomnilnik | 3 GB RAM DDR2 (800 MHz) |
| trdi disk | Seagate ST9100824AS (100 GB, 5400 rpm) |
| operacijski sistem | Microsoft Windows XP SP3 |
| omrežna povezava | 10 Mbit snemanje in nalaganje |

Tabela 2.1: Konfiguracija razvojnega sistema.

ASP.NET uporabili osnovne spletne tehnologije HTML, CSS in JS s knjižnico *jQuery*[1]. Pri analizi podatkov in izrisu vizualizacij smo poleg naštetega uporabili še vektorski slikovni format SVG, spletne vmesnike *Google Distance Matrix*[2], *Google Maps Geocoding*[3] in *Google Charts*[4] ter knjižnico D3[5].

Uporabniški vmesnik smo z zaledno aplikacijsko logiko povezali z uporabo komponente .NET-a za razvoj storitveno usmerjenih aplikacij WCF, arhitekturnega modela za porazdeljene sisteme REST, nabora tehnik za razvoj asinhronih spletnih aplikacij AJAX in tekstovnega standarda za izmenjavo podatkov JSON [10, 11].

Aplikacijo *OKG* smo v namen testiranja postavili na preprost spletni strežnik *Microsoft IIS Express*, ki je optimiziran za razvijalce in podpira vse glavne zmožnosti strežnika IIS 7 [12]. Pri strežniku se v namen diplomske naloge, razen osnovne postavitve aplikacije, nismo osredotočali na konfiguracijo. Za dostop do aplikacije smo uporabljali spletni brskalnik *Google Chrome*[6].

Med delom smo v namen spremljanja razvoja uporabili aplikacijo za izdelavo delovnih listov (angl. *worksheets*) *Microsoft Excel 2007*. Žični model spletne strani (angl. *wireframe*) na Sliki 3.4 v Poglavju 3.3 smo izdelali s spletno aplikacijo *Wireframe.cc*[7], entitetno-relacijske in druge diagrame, razvidne z nekaterih slik v diplomski nalogi, pa s spletno aplikacijo *LucidChart*[8]. Besedilo v žičnem modelu in diagramih je zaradi prostorske omejitve napisan v angleščini, saj so v večini primerov uporabljeni angleški izrazi krajši od istopomenskih slovenskih.

## 2.1 Programsko okolje Facebook Platform

Facebook od leta 2007 ponuja zunanjim razvijalcem dostop do svojih podatkov preko programskega okolja *Facebook Platform*, ki vključuje množico programskih vmesnikov in drugih orodij [13, 14]. Glavni komponenti okolja, ki smo se ju poslužili pri razvoju spletne aplikacije *OKG*, sta spletni vmesnik za programiranje *Graph API* in njegov sistem za

[1] Dostopno na naslovu: http://jquery.com
[2] Dokumentacija dostopna na naslovu: https://developers.google.com/maps/documentation/distancematrix
[3] Dokumentacija dostopna na naslovu: https://developers.google.com/maps/documentation/geocoding
[4] Dokumentacija dostopna na naslovu: https://developers.google.com/chart
[5] Dostopno na naslovu: http://d3js.org
[6] Dostopno na naslovu: https://www.google.com/intl/en/chrome/browser
[7] Dostopno na naslovu: https://wireframe.cc
[8] Dostopno na naslovu: https://www.lucidchart.com

avtentikacijo uporabnikov.

Graph API predstavlja jedro Facebook Platforme in je primarni način za programsko pridobivanje in objavo podatkov na Facebooku. Facebook na svoji spletni strani za razvijalce *Facebook Developers* objavlja dokumentacijo za njegovo uporabo in morebitne prihajajoče spremembe pri njegovem razvoju oz. vzdrževanju [15].

V namen razvoja naše aplikacije smo se osredotočili le na postopek avtentikacije (Poglavje 2.1.1) in na dostop do uporabniških podatkov (Poglavje 2.1.2) ter nismo raziskovali drugih funkcionalnosti Graph API-ja, kot je npr. objava vsebine na Facebooku.

### 2.1.1 Graph API: avtentikacija

Graph API za avtentikacijo uporablja odprti standard *OAuth 2.0*[9] in poleg osnovne avtentikacije podpira tudi uporabo dovoljenj (angl. *permissions*), s katerimi lahko od uporabnika zahtevamo dostop do določenih podatkov njegovega profila [15, 16]. Kot je razvidno z drugega koraka sledečega opisa uporabe avtentikacije preko Graph API-ja, zahtevana dovoljenja določimo ob izdelavi URL-ja za sprožitev Facebookovega dialoga za avtentikacijo. Spisek vseh uporabljenih dovoljenj v aplikaciji *OKG* je naveden v Tabeli 6 v dodatku.

Prvi korak implementacije uporabe avtentikacije preko Graph API-ja je izdelava preproste aplikacije v razdelku *Apps*[10] spletne strani Facebook Developers, v kateri registriramo spletno domeno in URL svoje spletne aplikacije, ki bo avtenticirala uporabnike [15]. Kot je razvidno s Slike 2.1, z izdelavo te aplikacije pridobimo aplikacijsko identifikacijsko številko (*App ID*) in aplikacijsko skrivnost (*App Secret*).

Pri drugem koraku, ki predstavlja implementacijo programske logike za avtentikacijo v naši spletni aplikaciji, Graph API podpira dve možnosti, in sicer implementacijo preko obličja z uporabo knjižnice *JavaScript Facebook SDK* ter preko zaledne strežniške kode [15]. Zaradi izbrane arhitekture aplikacije *OKG*, opisane v Poglavju 3.2, ki obličje loči od procesne aplikacijske logike, smo raziskali in uporabili način implementacije preko zaledja, ki smo ga opisali v nadaljevanju.

Ob določenem dogodku, npr. kliku uporabnika na gumb, na svoji spletni strani sprožimo preusmeritev na Facebookov dialog za avtentikacijo (Slika 2.2) z URL-jem (Vzorec kode 2.1), kateremu določimo poizvedbena parametra (angl. *query parameters*) [15]:

- `client_id` – identifikacijsko številko naše aplikacije na spletni strani Facebook Developers in

[9] Več informacij na naslovu: http://oauth.net/2

[10] Dostopno na naslovu: https://developers.facebook.com/apps

- `redirect_uri` – URL do naše spletne strani za preusmeritev uporabnika po končani avtentikaciji

ter po potrebi še neobvezne poizvedbene parametre:

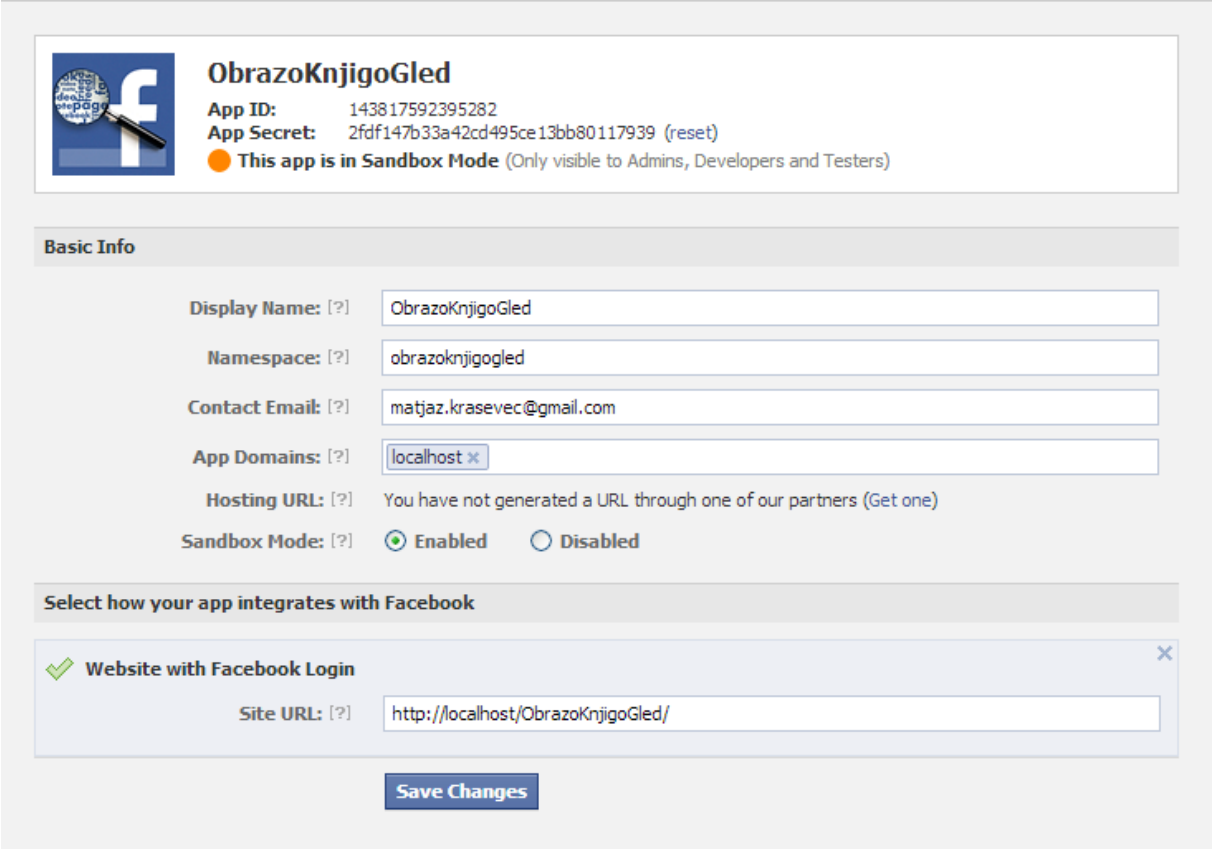
- `scope` – zahtevana dovoljenja, ločena z vejico,
- `state` – poljuben niz znakov za zavarovanje pred CSRF-ji,
- `response_type` – obliko Facebookovega odziva, z vrednostjo `code%20token`, `code` ali `token` ter
- `display` – obliko dialoga, z vrednostjo `page`, `popup`, `touch` ali `iframe`.

Kot je razvidno z diagrama na Sliki 2.3, ki prikazuje delovanje Facebookovega dialoga za avtentikacijo, se na tem mestu uporabnik s svojim uporabniškim imenom in geslom vpiše v Facebook ter zavrne ali dovoli naši aplikaciji uporabo svojih podatkov. V obeh primerih se uporabnikov spletni brskalnik vrne na našo spletno stran, in sicer na URL, ki smo ga določili v parametru `redirect_uri`, kateremu glede na uporabnikovo odločitev oz. morebitne napake s strani Facebooka in vrednost parametra `response_type` pripne ustrezne odzivne poizvedbene parametre (vzorca kode 2.2 in 2.3).

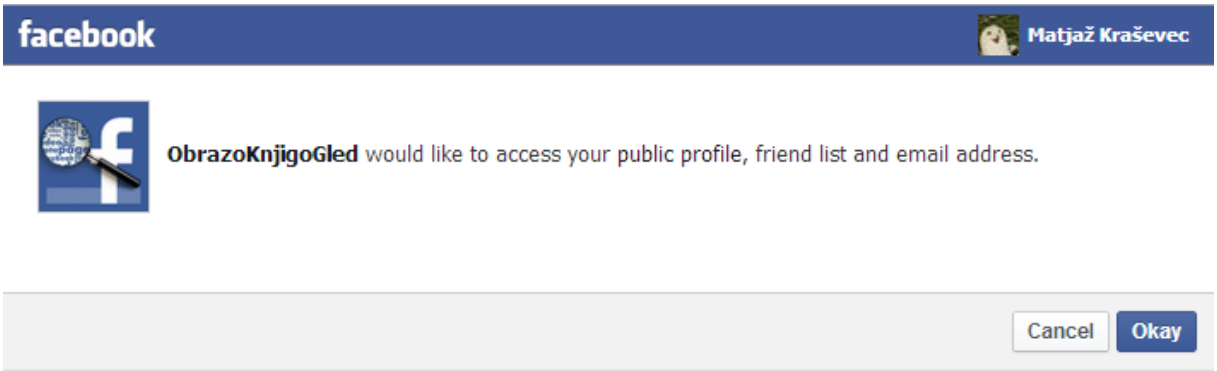
Če smo določili `response_type=code`, je nato potrebno vrednost odzivnega poizvedbenega parametra `code` preko Graph API-ja zamenjati za dostopni žeton (angl. *access token*). To storimo z zahtevo GET (Vzorec kode 2.4), v kateri poleg parametrov `client_id`, `redirect_uri` in `code` dodamo še `client_secret` – aplikacijsko skrivnost aplikacije, ki smo jo izdelali na spletni strani Facebook Developers. Odziv Graph API-ja na to zahtevo vsebuje v primeru napake razlog za napako, sicer pa dostopni žeton in njegov veljavnostni rok v obliki števila sekund do poteka veljavnosti žetona (Vzorec kode 2.5) [15].

Po prejetju dostopnega žetona je uporabnik efektivno avtenticiran v naši spletni aplikaciji in ta lahko preko Graph API-ja z dostopnim žetonom dostopa do Facebooka v imenu uporabnika. Po potrebi sledijo še shranitev dostopnega žetona na primerno mesto za kasnejšo uporabo v aplikaciji, spremljanje prijavnega stanja uporabnika in implementacija logike za odjavo uporabnika iz aplikacije. Facebook za to predlaga uporabo spremenljivke v seji spletnega brskalnika (angl. *session variable*) [15].

Pri raziskovanju načina za dostop podatkov, opisanega v sledečem poglavju, smo ugotovili, da Facebook aplikacijam z ustreznim dostopnim žetonom in dovoljenji preko Graph API-ja dovoli dostop le do podatkov profila uporabnika, kateremu dostopni žeton pripada, in profilov njegovih prijateljev, če to tudi oni sami dovolijo.




Slika 2.1: Aplikacija na spletni strani Facebook Developers, preko katere aplikacija *OKG* avtenticira uporabnike in dostopa do podatkov na Facebooku. Razvidna sta aplikacijska identifikacijska številka, aplikacijska skrivnost in osnovne nastavitve, primerne za lokalni razvoj in testiranje aplikacije.




Slika 2.2: Facebookov dialog za avtentikacijo, sprožen z URL-jem, razvidnim iz Vzorca kode 2.1. S slike je razvidno, da želi aplikacija *ObrazoKnjigoGled* dostop do javnih podatkov, seznama prijateljev in elektronskega naslova našega profila.

```
https://www.facebook.com/dialog/oauth?
    client_id=143817592395282
   &redirect_uri=http%3A%2F%2Flocalhost%2FOKG%2FFacebookCheck.aspx
   &scope=email
   &state=__okg_login
   &response_type=code
   &display=popup
```

Vzorec kode 2.1: Primer URL-ja za sprožitev Facebookovega dialoga za avtentikacijo.

```
http://localhost/OKG/FacebookCheck.aspx?
    error_reason=user_denied
   &error=access_denied
   &error_description=The+user+denied+your+request.
```

Vzorec kode 2.2: Odzivni URL Facebookovega dialoga za avtentikacijo, sproženega z URL-jem, razvidnim iz Vzorca kode 2.1, v primeru uporabnikovega klika na gumb za preklic (*Cancel*).

```
http://localhost/OKG/FacebookCheck.aspx?
    code=AQDQiO_MgSOaBkoyPDO_aVsMAh9dOU3-oNL8Yo1jRL4ImmwvmF6G5l ...
   &state=__okg_login
```

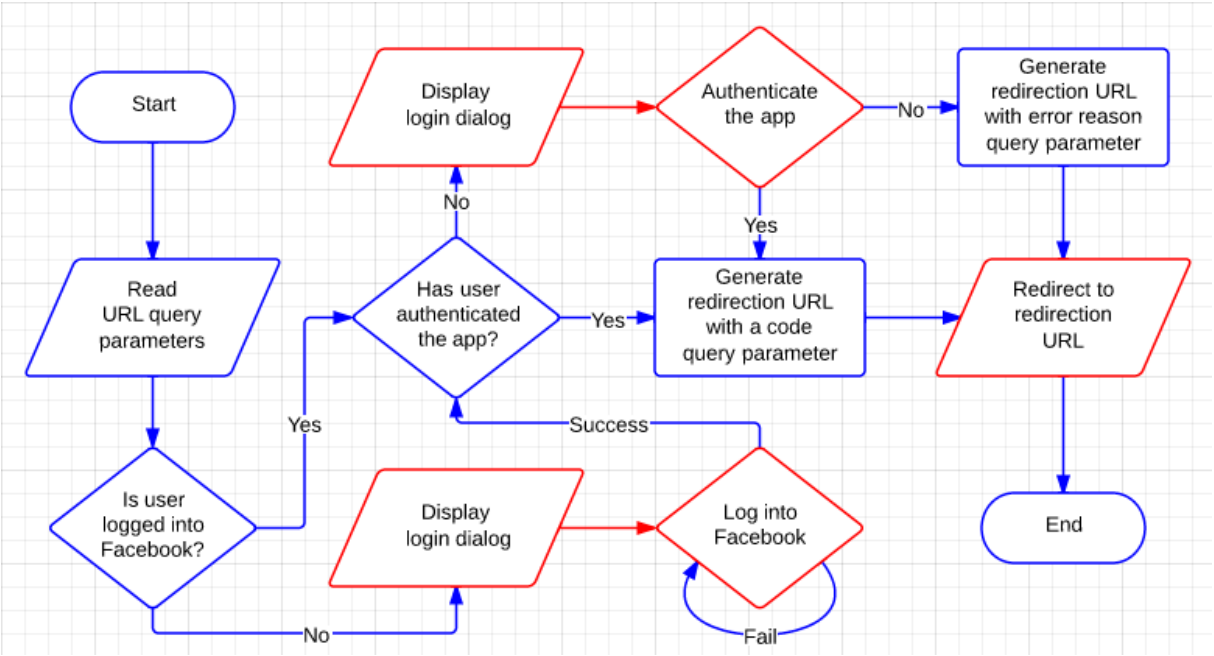
Vzorec kode 2.3: Odzivni URL Facebookovega dialoga za avtentikacijo, sproženega z URL-jem, razvidim iz Vzorca kode 2.1, v primeru uporabnikovega klika na gumb za nadaljevanje (*Okay*).

```
GET https://graph.facebook.com/oauth/access_token?
    client_id=143817592395282
   &redirect_uri=http%3A%2F%2Flocalhost%2FOKG%2FFacebookCheck.aspx
   &client_secret=2fdf147b33a42cd495ce13bb80117939
   &code=AQDQiO_MgSOaBkoyPDO_aVsMAh9dOU3-oNL8Yo1jRL4ImmwvmF6G5l ...
```

Vzorec kode 2.4: Zahteva GET za pridobitev dostopnega žetona iz vrednosti `code`, dobljene v odzivu Facebookovega dialoga za avtentikacijo (Vzorec kode 2.3).

```
access_token=CAACCzSSB1hIBAFWFf4ZAdFSbZBAzhNvktrwlCZBinoaP90CSe ...
&expires=5183911
```

Vzorec kode 2.5: Primer uspešnega Facebookovega odziva na zahtevo GET za pridobitev dostopnega žetona, razvidno iz Vzorca kode 2.4.




Slika 2.3: Entitetno-relacijski diagram delovanja Facebookovega dialoga za avtentikacijo. Rdeče obarvane entitete se izvajajo na odjemalcu, modro obarvane pa na Facebookovem strežniku.

### 2.1.2 Graph API: dostop do podatkov

Podatki so na Facebooku shranjeni v objektih Facebookovega socialnega grafa, ki vsebujejo unikatne identifikacijske številke, se delijo na več tipov, vsebujejo polja s preprostimi ali kompleksnimi podatki ter so medsebojno povezani [17]. Polja in povezave posameznih objektov, ki smo jih potrebovali pri razvoju aplikacije *OKG*, smo navedli v dodatku [15].

Omembe vredno je še, da so vsi datumi v podatkih Graph API-ja predstavljeni v obliki zapisa, definiranega v mednarodnem standardu *ISO-8601*, z datumom, urami, minutami, sekundami in odmikom od UTC-ja [18]. Graph API v svojih odzivih pri poljih, ki vsebujejo seznam podatkov, uporablja ostranjevanje (angl. *paging*), kar pomeni, da podatke v seznamu razdeli na več strani, na vsaki strani pa vključi polje `paging` s podpoljema `next` in `previous`, ki vsebujeta povezave do naslednje oz. prejšnje strani v seznamu [15].

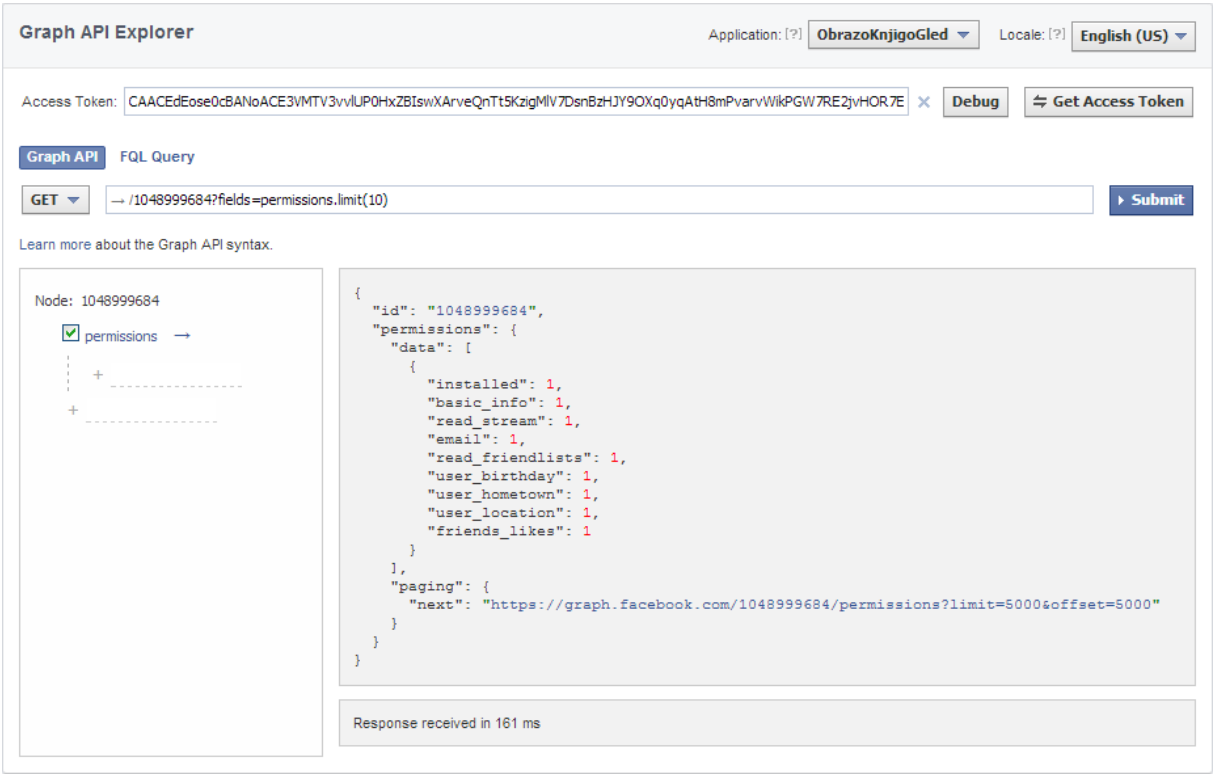
Dostop do objektov Facebookovega socialnega grafa preko Graph API-ja temelji na uporabi modela REST oz. specifičneje, na uporabi zahtev GET, katerim določimo [15]:

- Graph API-jev URL, in sicer `https://graph.facebook.com/`,
- dostopni žeton, na katerega so vezana ustrezna dovoljenja za dostop do podatkov,

- identifikacijsko številko želenega objekta,
- opcijski modifikator `fields` z zahtevanimi polji in povezavami. Pri določanju povezav je možno tudi gnezdenje dodatnih modifikatorjev, vezanih nanje. V primeru ostranjevanja lahko z gnezdenim modifikatorjem `limit` določimo število podatkov na posamezni strani.

Graph API se na zahteve odzove s podatki v obliki JSON. Primera zahteve GET in odziva Graph API-ja nanjo sta razvidna iz vzorcev kode 2.6 in 2.7.

Opazili smo, da je Facebookova dokumentacija Graph API-ja pri tematikah, ki opisujejo strukturo objektov Facebookovega socialnega grafa, površna in zastarela, zatorej se le nanjo med raziskavanjem odzivov Graph API-ja nismo mogli zanašati. Informacije v omenjeni dokumentaciji smo preverili z uporabo orodja Raziskovalec Graph API-ja (angl. *Graph API Explorer*)[11], ki temelji na preprostem odjemalcu REST in omogoča testiranje Graph API-ja s prikazom njegovih odzivov (Slika 2.4).




Slika 2.4: Raziskovalec Graph API-ja. S slike je razvidna zahteva GET za prikaz pridobljenih dovoljenj, vezanih na trenutni dostopni žeton, in Graph API-jev odziv nanjo.

[11] Dostopno na naslovu: https://developers.facebook.com/tools/explorer

```
GET https://graph.facebook.com/1048999684?
    fields=id,name,hometown,updated_time,likes
        .limit(2)
        .fields(id,name,category,created_time)
   &access_token=CAACEdEose0cBALtpE8KZBePaZBKjRmkrrAJ45JQOEryC8 ...
```

Vzorec kode 2.6: Primer zahteve GET za pridobitev podatkov preko Graph API-ja. Zahteva vsebuje pot do Graph API-ja, identifikacijsko številko objekta (`1048999684`), katerega podatke želimo, modifikator `fields` z definicijo želenih polj in povezav tega objekta, gnezdena modifikatorja `.fields` in `.limit` ter dostopni žeton (`access_token`).

```
{
  "id": "1048999684",
  "name": "Matjaž Kraševec",
  "hometown": {
    "id": "108616509162456",
    "name": "Novo Mesto"
  },
  "updated_time": "2013-07-06T11:15:18+0000",
  "likes": {
    "data": [{
        "id": "230995986911532",
        "name": "Minutephysics",
        "category": "Science website",
        "created_time": "2013-07-11T17:20:57+0000"
      }, {
        "id": "103673816367868",
        "name": "Fakulteta za računalništvo in informatiko",
        "category": "Education",
        "created_time": "2011-10-23T03:03:42+0000"
      }
    ],
    "paging": {
      "next": https://graph.facebook.com/1048999684/likes?
              limit=2
             &fields=id,name,category&offset=2
             &__after_id=230995986911532
    }
  }
}
```

Vzorec kode 2.7: Graph API-jaev odziv na zahtevo GET, razvidno iz Vzorca kode 2.6. Gre za objekt `user`, ki vsebuje preprosta polja `id`, `name` in `updated_time`, kompleksno polje `hometown` ter povezavo `likes`. Slednja je zaradi ostranjevanja razdeljena na več strani, do katerih lahko dostopamo s povezavami v polju `paging`.

# Poglavje 3

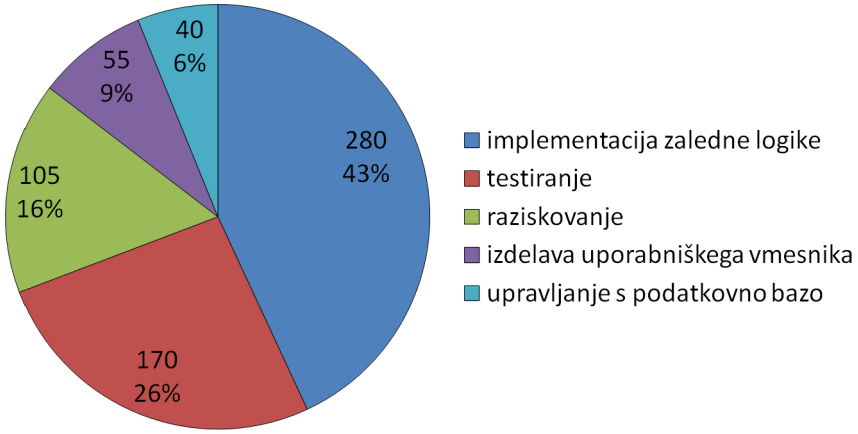
# Razvoj in delovanje aplikacije

V sledečih poglavjih smo podrobno opisali potek razvoja in delovanje aplikacije *ObrazoKnjigoGled*. V Poglavju 3.1 smo predstavili razdelitev razvojnega problema na manjše dele in naš način vodenja razvoja, kar smo nadaljevali z opisom izbrane aplikacijske arhitekture v Poglavju 3.2, izdelavo uporabniškega vmesnika v Poglavju 3.3, implementacijo sistema za avtentikacijo uporabnikov v Poglavju 3.4) in implementacijo logike za prenos uporabniških podatkov v Poglavju 3.5. V Poglavju 3.6 smo opisali knjižnici za generacijo vizualizacij in geokodiranje ter navedli vsebino analitičnega poročila o uporabniških podatkih.

## 3.1 Projektno vodenje razvoja

Za boljši pregled nad razvojem aplikacije smo razvojni problem razdelili na več glavnih nalog s podnalogami. Glavne naloge so bile:

- raziskava delovanja in uporabe Facebook Platforme, vključno z avtentikacijo, uporabo dovoljenj in programskim dostopom do podatkov na Facebooku preko Graph API-ja (Poglavje 2.1),
- določitev aplikacijske arhitekture (Poglavje 3.2),
- implementacija uporabniškega vmesnika (Poglavje 3.3),
- razčlenitev podatkov Facebookovega socialnega grafa v ustrezne objekte za preslikavo v relacijske tabele in izdelava lokalne podatkovne baze za hranitev uporabniških podatkov (Poglavje 3.5.1),




Slika 3.1: Krožni grafikon deležev časa, porabljenega za izdelavo aplikacije.

- implementacija aplikacijske logike za prenos podatkov s Facebooka v lokalno podatkovno bazo in optimizacija te logike (Poglavje 3.5.2),
- implementacija aplikacijske logike za analizo uporabniških podatkov in generacijo poročila z vizualizacijami (Poglavje 3.6),
- implementacija podpornih knjižnic za razvoj in delovanje aplikacije, vključno s knjižnicami za sledenje napak pri delovanju, časovno meritev izvajanja algoritmov, avtentikacijo uporabnikov (Poglavje 3.4), geokodiranje (Poglavje 3.6.2), preprostejšo uporabo dinamičnega programiranja in refleksije ter pomoč pri zaledni izdelavi vizualizacij (Poglavje 3.6.1) in drugih elementov obličja aplikacije,
- testiranje in razhroščevanje aplikacije.

Naloge smo pred oz. med razvojem vnesli v preglednico in pri vsaki izmed njih zapisali opis naloge, datum vnosa v tabelo, datuma začetka in konca reševanja naloge ter število porabljenih ur, pri čemer smo ločevali ure za raziskovanje določene tematike, upravljanje s podatkovno bazo, izdelovanje uporabniškega vmesnika, implementacijo zaledne logike in testiranje aplikacije.

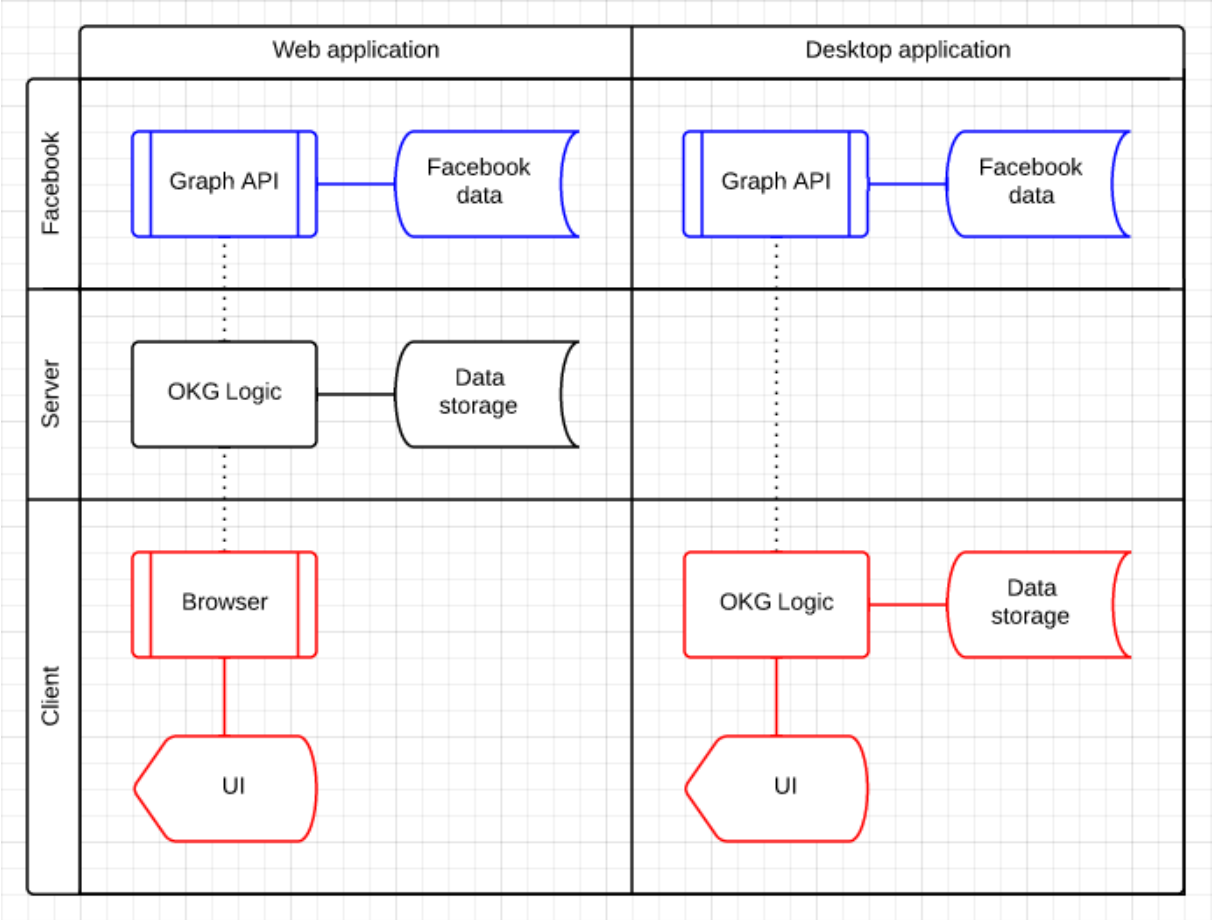
Celotno aplikacijo *OKG* s pripadajočimi knjižnicami smo razvili v približno 650 urah, kar predstavlja skoraj 3 mesece in pol rednega dela. Kot je razvidno s krožnega grafikona na Sliki 3.1, smo največji delež tega časa porabili za implementacijo zaledne logike (43,08% oz. 280 ur), testiranje (26,15% oz. 170 ur) in raziskovanje (16,15% oz. 105 ur). Nekoliko manjši delež porabljenega časa sta predstavljali izdelava uporabniškega vmesnika (8,46% oz. 55 ur) in upravljanje s podatkovno bazo (6,15% oz. 40 ur).

## 3.2 Aplikacijska arhitektura

V sledečih poglavjih smo opisali izbrano arhitekturo aplikacije *ObrazoKnjigoGled*. V prvem delu smo opisali zunanjo strukturo oz. model porazdelitve delovne obremenitve in virov aplikacije, v drugem delu pa smo se osredotočili na notranjo strukturo oz. arhitekturo aplikacije glede na njene komponente, ki smo jih morali implementirati sami.

### 3.2.1 Zunanja struktura aplikacije

Odločiti smo se morali, ali bomo razvili spletno, ali namizno aplikacijo. Izrisali smo strukturna modela za obe možnosti, ki sta razvidna s Slike 3.2. Pomembnejše razlike med njima smo navedli v Tabeli 3.1. Pri tem smo upoštevali, da namizni model pri delovanju poleg Facebooka ne uporablja nobenega drugega spletnega strežnika. Možnosti




Slika 3.2: Strukturna modela spletne (levo) in namizne (desno) aplikacije *OKG*. Entiteta *OKG Logic* prestavlja zaledno logiko, entiteta *UI* pa uporabniški vmesnik aplikacije.

| | namizna aplikacija | spletna aplikacija |
|---|---|---|
| izvajanje, poraba virov in dostop do aplikacije | odjemalec aplikacijo požene in do nje dostopa lokalno; aplikacija za delovanje uporablja le vire odjemalca | aplikacija se izvaja na strežniku in odjemalec do nje dostopa preko spleta; aplikacija za delovanje obličja uporablja vire odjemalca, za delovanje zaledja pa vire strežnika |
| obličje | namizni grafični uporabniški vmesnik, implementiran z Windows Forms, WPF ali drugo podobno tehnologijo | spletni uporabniški vmesnik, implementiran s HTML, CSS in JS oz. drugimi programskimi jeziki, ki so podprti v spletnih brskalnikih |
| dostop do Facebooka | odjemalec sam za avtentikacijo in dostop do podatkov; preko spleta | odjemalec in strežnik za avtentikacijo ter strežnik sam za dostop do podatkov; preko spleta |
| skladiščenje podatkov | odjemalec hrani podatke svojih lokalnih uporabnikov | strežnik hrani podatke vseh uporabnikov |
| nadgrajevanje aplikacije | odjemalec potrebuje novejšo različico aplikacije; potrebna je implementacija podpore nagrajevanju in uporabi podatkov starejših različic aplikacije | aplikacijo nadgradimo na strežniku in je odjemalcu dostopna takoj |
| kontrola nad uporabniki in izvajanjem aplikacije | težja; delovanje se zapisuje v datoteko na odjemalcu, ki nam jo uporabnik posreduje po končani uporabi aplikacije | lažja; delovanje se zapisuje v datoteko na strežniku, ki nam je dostopna takoj |
| možnost analize podatkov več uporabnikov hkrati | ne; podatki uporabnika se nahajajo na odjemalcu in nam niso na voljo | da; podatki vseh uporabnikov se nahajajo na strežniku in so nam vedno na voljo |

Tabela 3.1: Razlike med modeloma namizne in spletne aplikacije.

za implementacijo spletne aplikacije v oblaku nismo raziskovali.

Odločili smo se za centralizirano spletno rešitev, saj med drugim omogoča zbiranje in posledično možnost analiziranja vseh informacij na enem mestu, tj. na strežniku, ter boljšo kontrolo nad izvajanjem aplikacije, saj se ne glede na število uporabnikov izvaja le ena instanca zaledne logike. Edina slabost spletne rešitve, ki smo jo opazili, je velika poraba računalniških virov na strežniku, saj bi se le-ta v primeru namizne rešitve bolje porazdelila po računalnikih uporabnikov.

Izbrani strukturni model aplikacije *OKG* predstavlja dvojni model *odjemalec-strežnik*[1], kjer je v prvem odjemalec uporabnik in strežnik naš računalnik, v drugem pa odjemalec naš računalnik in strežnik Facebook.

### 3.2.2 Notranja struktura aplikacije

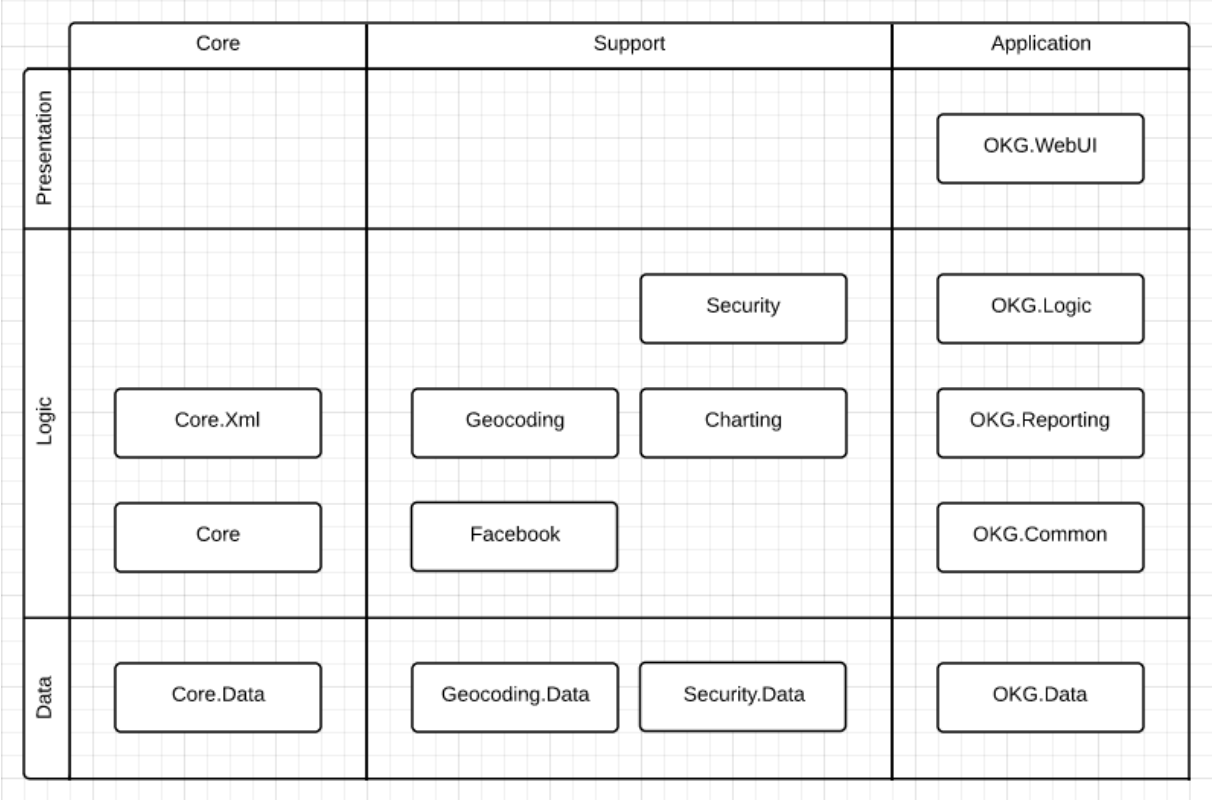
Aplikacijo *OKG* smo med razvojem strukturno razdelili na več projektnih knjižnic, ki smo jih po funkcionalnosti in uporabi združili v horizontalne sklope in vertikalne sloje (Slika 3.3) ter jih ustrezno referencirali med seboj.

Pri horizontalni delitvi projektnih knjižnic smo poskušali čim bolje upoštevati princip programiranja DRY, katerega namen je zmanjševanje ponavljanja kode [19]. Metode in objekte smo definirali le enkrat in jih ustrezno referenčno uporabili na več lokacijah v kodi. S tem smo pridobili na obvladljivosti kode, saj so morebitne manjše spremembe potrebne le na enem mestu in imajo predvidljive posledice na delovanje aplikacije. Z upoštevanjem principa DRY smo si zagotovili možnost ponovne uporabe delov kode pri nadgrajevanju aplikacije *OKG* in razvoju drugih aplikacij. Horizontalne sklope v aplikaciji *OKG* predstavljajo:

- knjižnice *Core*, v katerih smo definirali preproste metode in objekte, ki niso vezani na specifike aplikacije *OKG* in bi prišli v poštev za ponovno uporabo pri razvoju drugih aplikacij;

- knjižnice, namenjene bolj specifičnim funkcionalnostim in podpori pri programiranju, vključno s knjižnicami za: uporabo Graph API-ja (*Facebook*), generiranje vizualizacij (*Charting*), avtentikacijo uporabnikov (*Security*) in geokodiranje (*Geocoding*);

[1] Več informacij na naslovu: http://en.wikipedia.org/wiki/Client%E2%80%93server_model




Slika 3.3: Projektne knjižnice aplikacije *OKG*, razporejene horizontalno v sklope in vertikalno v sloje. Vsaka knjižnica ima dostop do knjižnic, ki so nižje in/ali bolj levo od nje (npr. knjižnica *OKG.WebUI* ima dostop do vseh drugih knjižnic, knjižnica *Core.Data* pa do nobene).

- knjižnice, specifične za aplikacijo *OKG*, vključno s spletnim uporabniškim vmesnikom (*OKG.WebUI*), procesno logiko (*OKG.Logic*), logiko za dostop do podatkov (*OKG.Data*) in knjižnicama za upravljanje s konfiguracijo in sledenjem aplikacije (*OKG.Common*) ter za analizo podatov in izdelavo poročila (*OKG.Reporting*).

Z uporabo trislojne aplikacijske arhitekture, smo vertikalno ločili predstavitveno logiko od procesne in podatkovne. Ta arhitektura je enosmerno linearna, kar pomeni, da lahko predstavitveni sloj neposredno dostopa le do aplikacijskega, aplikacijski pa do podatkovnega. Večslojnost je pripomogla k izboljšanju preglednosti in obvladljivosti kode, saj pri morebitni spremembi v enem sloju ni nujno potrebno spremeniti tudi drugih slojev. Trislojna aplikacijska arhitektura vključuje [20, 21]:

- predstavitveni sloj, ki vključuje uporabniški vmesnik. Ta uporabniku podaja statično oz. dinamično vsebino in mu omogoča pošiljanje zahtev procesni logiki;

- procesni sloj, ki vsebuje logiko za obravnavo zahtev s predstavitvenega sloja in generacijo dinamične vsebine, pri čemer po potrebi podatke shranjuje na oz. pridobiva s podatkovnega sloja;

- podatkovni sloj, ki vključuje sistem za upravljanje s podatkovnimi bazami in logiko, v kateri so definirane metode in objekti, preko katerih procesni sloj dostopa do omenjenega sistema. V primeru aplikacije *OKG* ima ta logika poudarek na LINQ-to-SQL-ovih objektih `DataContext`.

## 3.3 Uporabniški vmesnik

Pri razvoju obličja aplikacije *OKG* smo se odločili za izdelavo vmesnika SPI – asinhronega uporabniškega vmesnika, ki temelji na eni sami spletni strani in ponuja uporabnikom tekočo uporabniško izkušnjo, podobno namiznim aplikacijam [22]. Ta odločitev je temeljila na naši osebni želji po znanju razvoja takšnega vmesnika.

Izdelali smo žični model, ki prikazuje postavitev elementov na spletni strani (Slika 3.4). Kot je razvidno z modela, smo spletno stran razdelili na več sekcij, in sicer:

- glavo z naslovom aplikacije, glavnim navigacijskim menijem in prostorom za prikaz sporočil uporabniku,

- Stran za vpis (angl. *Login screen*) s pozdravnim besedilom in gumbom za vpis v aplikacijo,

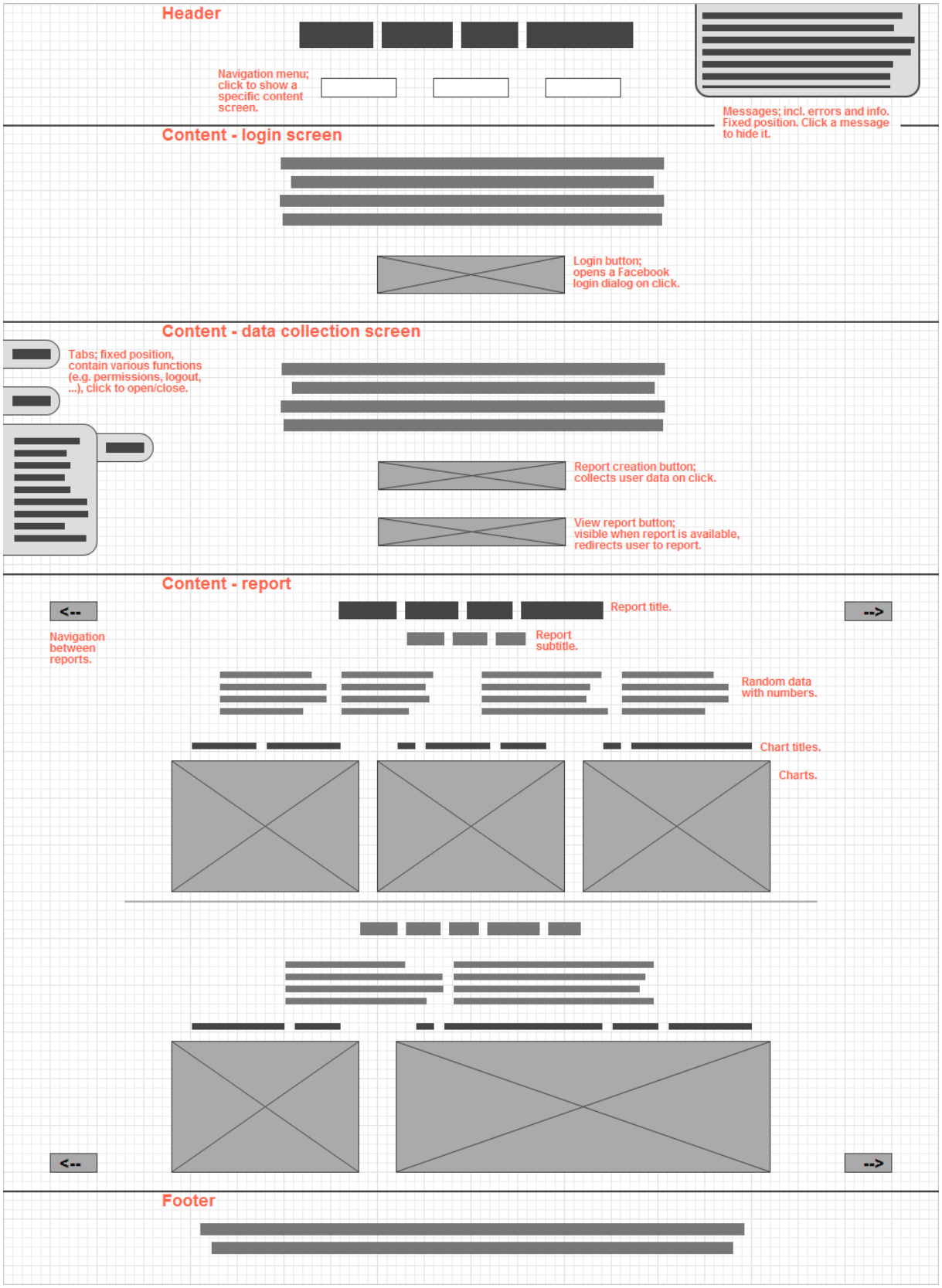
- Stran za prenos podatkov (angl. *Data collection screen*) s kratkimi navodili za uporabo, gumboma za generacijo in ogled poročila ter zavihki z raznimi funkcijami, kot sta upravljanje z dovoljenji in izpis iz aplikacije,

- Poročilo (angl. *Report*) s tekstovnimi informacijami, vizualizacijami in gumbi za navigacijo med sekcijami poročila (Poglavje 3.6.3) in

- nogo z navedbo o licenci, pod katero smo objavili aplikacijo.

Glava in noga sta uporabniku vidni vedno, izmed drugih, vsebinsko bolj specifičnih, sekcij, pa mu je lahko naenkrat vidna le ena.

Ker vmesniki SPI ne delujejo na principu vračanja vsebine (angl. *postback*) [22, 23], smo morali najti tehnični pristop, ki omogoča spletnemu brskalniku ohranitev spletne strani medtem, ko le-ta potrebuje komunikacijo s strežnikom.




Slika 3.4: Žični model spletne strani aplikacije *OKG*.

Uporabili smo pristop, ki temelji na naboru tehnik za razvoj asinhronih aplikacij AJAX in tekstovnemu standardu za izmenjavo podatkov JSON. Naša implementacija v principu deluje tako, da uporabniški vmesnik preko metod, implementiranih v jeziku JS s pomočjo knjižnice jQuery, asinhrono pošilja zahteve na strežnik. Strežnikova aplikacijska logika zahteve obdela in uporabniškemu vmesniku vrne rezultat oz. dinamično vsebino v podatkovni obliki JSON. Logika JS na strani uporabniškega vmesnika podatke nato ustrezno obdela: če gre za preproste rezultate jih uporabi v izračunih, če pa gre za vsebino, v našem primeru zakoridano kodo HTML in/ali JS, jo ustrezno postavi, prikaže oz. izvede na spletni strani uporabniškega vmesnika.

V namen generiranja statične in dinamične vsebine uporabniškega vmesnika smo na datotečnem sistemu strežnika izdelali datoteke za prikaz in delovanje spletne strani, ki vsebujejo statične elemente, vključno s prikaznimi objekti (značke HTML), dizajnom (slogi CSS) in funkcionalnostjo (koda JS), ter implementirali projektno knjižnico *Core.Xml*, v kateri smo v kodi C# definirali vmesnike oz. objekte, ki se dinamično serializirajo v HTML, CSS in JS. Primer uporabe te knjižnice je razviden iz vzorcev kode 2, 1 in 3 v dodatku.

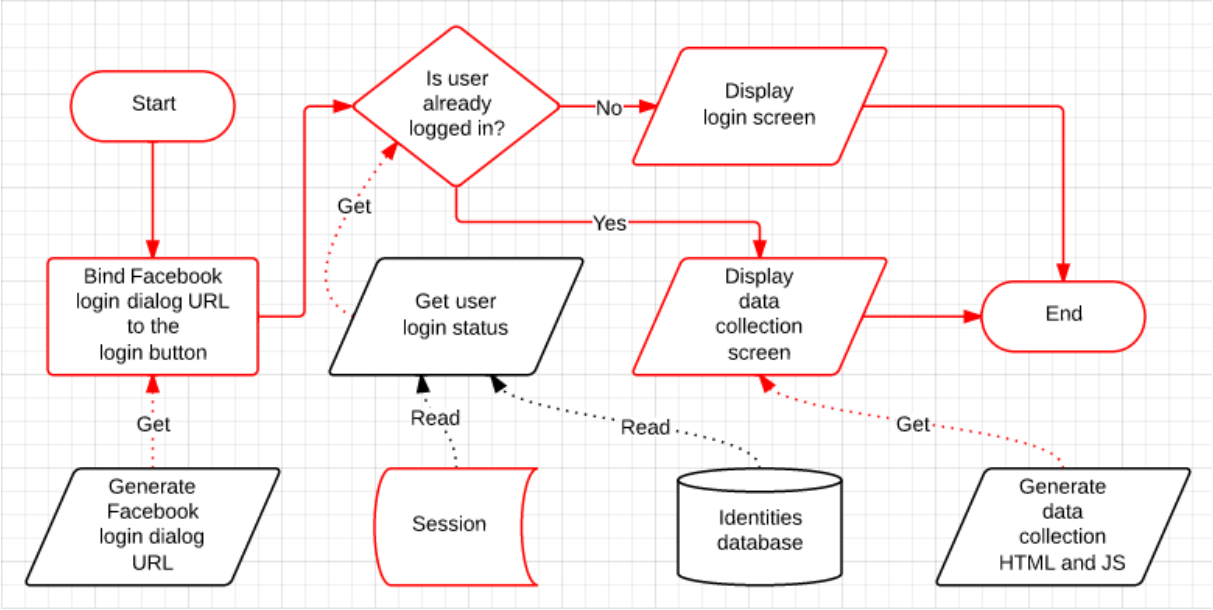
Pri izdelavi spletne strani smo uporabili preprost in konsistenten dizajn, na katerega se v namen diplomske naloge nismo posebno osredotočali. Aplikacijo smo med razvojem testirali v spletnem brskalniku Google Chrome in se nismo ozirali na optimizacijo uporabniškega vmesnika za druge brskalnike. Končni grafični videz spletne strani aplikacije *OKG* je razviden s slik v Poglavju 4.1 in dodatku.

### 3.3.1 Inicializacija uporabniškega vmesnika

Ob povezavi uporabnika oz. odjemalca z aplikacijo *OKG* se inicializira uporabniški vmesnik, kot je razvidno z diagrama na Sliki 3.5. V začetku se generira URL za sprožitev Facebookovega dialoga za avtentikacijo (Poglavje 2.1.1) in pripne (angl. *bind*) na dogodek ob kliku gumba za vpis na Strani za vpis.

Aplikacija nato preveri, ali je uporabnik že vpisan; pogleda, ali uporabnikova seja vsebuje uporabniški identitetni objekt, in v primeru, da ga, če je ta objekt veljaven oz. če obstajajo njegovi podatki tudi v podatkovni bazi uporabniških identitet aplikacije. Več o sami avtentikaciji uporabnikov smo zapisali v Poglavju 3.4.

Če uporabnik še ni vpisan, se mu prikaže Stran za vpis. Vsebina te strani je statična in dostopna že takoj po inicializaciji uporabniškega vmesnika, torej zanjo ni potrebna dinamična generacija vsebine. Vsa ostala vsebina spletne strani, omenjena v nadaljevanju,




Slika 3.5: Entitetno-relacijski diagram inicializacije uporabniškega vmesnika. Rdeče obarvane entitete se izvajajo oz. nahajajo na odjemalcu, črno obarvane pa na strežniku. Pikčaste puščice z napisom *Get* predstavljajo zahteve GET z uporabno tehnik AJAX.

se generira dinamično in pripne na obstoječo spletno stran med izvajanjem aplikacije.

Če je uporabnik že vpisan, spletna aplikacija generira vsebino Strani za prenos podatkov, jo pošlje uporabniškemu vmesniku, ta pa jo ustrezno obdela in prikaže.

## 3.4 Avtentikacija uporabnikov

Registracijo in avtentikacijo uporabnikov aplikacije *OKG* smo implementirali v projektni knjižnici *Security* z uporabo lokalne podatkovne baze in Facebookovega sistema za avtentikacijo preko Graph API-ja (Poglavje 2.1.1).

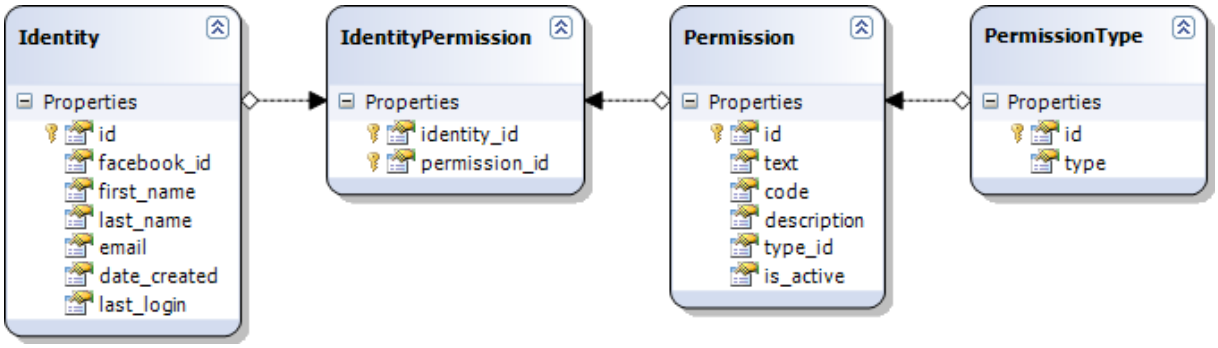
Izdelali smo aplikacijo na spletni strani Facebook Developers in ji določili domeno in URL aplikacije *OKG* (Slika 2.1 v Poglavju 2.1.1), s čimer smo pridobili aplikacijsko identifikacijsko številko in aplikacijsko skrivnost. Z naprednimi nastavitvami aplikacije na spletni strani Facebook Developers se v namen diplomskega dela nismo ukvarjali.

Za potrebe knjižnice *Security* smo izdelali podatkovno bazo, katere struktura je razvidna z diagrama na Sliki 3.6. Definirali smo glavni tabeli `Identity`, ki vsebuje uporabniške identitetne podatke, in `Permission`, ki vsebuje dovoljenja za uporabo z Graph API-jem, ter ju ustrezno povezali.

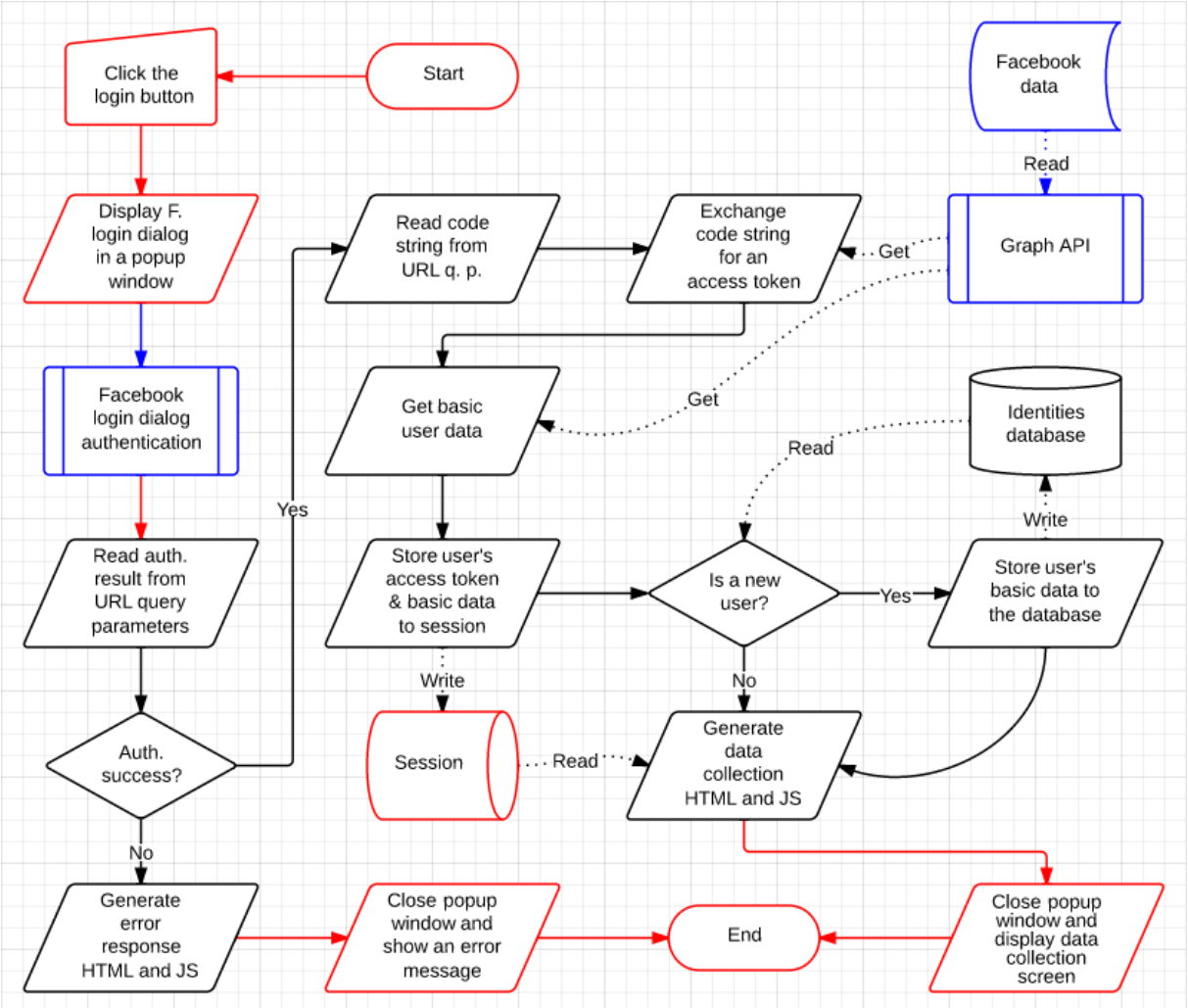
Za funkcionalnost Strani za vpis na uporabniškem vmesniku aplikacije *OKG* smo implementirali kodo, katere delovanje je razvidno z diagrama na Sliki 3.7. Na strani ima uporabnik možnost klika na gumb za vpis, ob čimer se v pojavnem oknu prikaže Facebookov dialog za avtentikacijo, preko katerega se uporabnik avtenticira, kot je opisano v Poglavju 2.1.1. Po končani avtentikaciji omenjeni dialog aplikaciji *OKG* pošlje odgovor v obliki URL-ja, iz katerega naša aplikacija ugotovi ali je bila avtentikacija uspešna. Zaledna aplikacijska logika nato zgenerira dinamično vsebino, s katero uporabniški vmesnik zapre pojavno okno in izpiše sporočilo o uspešnosti avtentikacije.

V primeru uspešne avtentikacije aplikacija *OKG* prejme vrednost `code`, katero preko Graph API-ja zamenja za dostopni žeton. Nato Graph API-ju pošlje zahtevo po uporabnikovi elektronski pošti ter njegovih osnovnih in javnih podatkih, vključno z uporabniško identifikacijsko številko, imenom, priimkom in spolom. Omenjeni podatki se skupaj z dostopnim žetonom v obliki uporabniškega identitetnega objekta zapišejo v sejo spletnega brskalnika uporabnika. Če gre za uporabnikovo prvo prijavo v aplikacijo, se njegovi identitetni podatki shranijo še v lokalno podatkovno bazo knjižnice *Security*. Ob odjavi uporabnika iz aplikacije se njegovi identitetni podatki pobrišejo iz seje.

Implementirali smo še uporabo dovoljenj za dostop do drugih uporabniških podatkov, ki smo jih navedli v Tabeli 6 v dodatku. Zahteve po teh dovoljenjih sicer delujejo po isti logiki kot osnovna avtentikacija, vendar smo jih z vidika uporabnika izvedli ločeno od nje; uporabnikom ni treba privoliti k vsem zahtevam po dovoljenjih že pri prvem vpisu v aplikacijo, ampak si lahko po vpisu na Strani za prenos podatkov sami izberejo, katere podatke bodo aplikaciji dovolili uporabiti.




Slika 3.6: Diagram tabel in njihovih medsebojnih povezav v podatkovni bazi projektne knjižnice za registracijo in avtentikacijo uporabnikov.




Slika 3.7: Entitetno-relacijski diagram delovanja Strani za vpis. Rdeče obarvane entitete se izvajajo oz. nahajajo na odjemalcu, modro obarvane na Facebookovem strežniku, črno obarvane pa na našem strežniku. Pikčaste puščice z napisom Get predstavljajo zahteve GET z uporabno tehnik AJAX.

## 3.5 Prenos uporabniških podatkov

Po implementaciji avtentikacijskega sistema je naslednji korak razvoja aplikacije *ObrazoKnjigoGled* predstavljala implementacija logike za prenos podatkov z uporabniških profilov na Facebooku v lokalno podatkovno bazo. Izziva smo se lotili z izdelavo podatkovne baze (Poglavje 3.5.1) in nadaljevali z implementacijo procesne logike (Poglavje 3.5.2).

### 3.5.1 Podatkovna baza

Pri izdelavi podatkovne baze za hranitev uporabniških podatkov s Facebooka smo poskušali čim bolje posnemati strukturo podatkov v odzivih Graph API-ja in se nismo ozirali na tehnike za optimizacijo baz, kot je npr. normalizacija. Izdelali smo relacijske tabele za vse potrebne objekte Facebookovega socialnega grafa in vsaki izmed njih dodali stolpce, ki predstavljajo preprosta polja teh objektov, za kompleksna polja in povezave pa smo izdelali nove tabele z ustreznimi relacijami. Polja in povezave objektov Facebookovega socialnega grafa, ki smo jih vključili v podatkovno bazo, smo navedli v tabelah v dodatku. Relacijski tabeli `fbUser` smo dodali še stolpec `custom_date_collected`, katerega vrednosti povedo, kdaj so bili podatki posameznega uporabnika nazadnje preneseni s Facebooka.

Poleg relacijskih tabel objektov smo v podatkovni bazi izdelali še Preslikovalno tabelo, v kateri smo definirali preslikave med relacijami v podatkovni bazi in polji v odzivnih podatkih Graph API-ja. V vsaki vrstici te tabele smo za določeno relacijo vpisali sledeče vrednosti:

- `sql_entity` – ime nadrejene relacijske tabele, ki predstavlja objekt Facebookovega socialnega grafa,
- `sql_field` – ime podrejene relacijske tabele, ki predstavlja kompleksno polje ali povezavo objekta Facebookovega socialnega grafa,
- `api_field` – ime polja v podatkih Graph API-ja, kateremu relacija pripada,
- `api_modifier` – modifikatorji za uporabo v zahtevah GET, ki jih pošiljamo Graph API-ju,
- `is_user_only` – zastavica, ki pove, ali relacija poleg pri uporabniku samem pride v poštev tudi pri njegovih prijateljih,

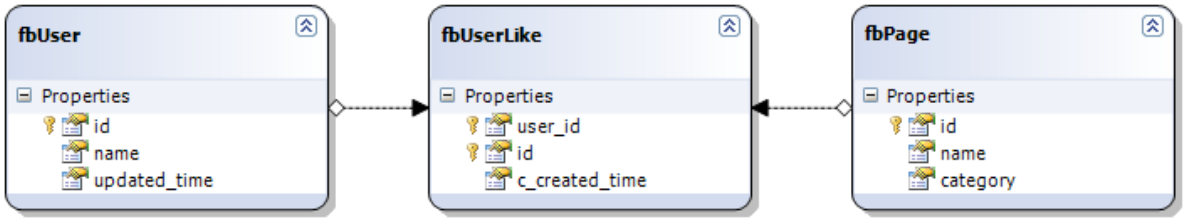
- `is_connection` – zastavica, ki pove, ali gre za relacijo s kompleksnim poljem ali za povezavo oz. ali polje v podatkih Graph API-ja vsebuje vmesni element `data`,
- `is_active_reference` – zastavica, ki pove, ali naj se prenesejo tudi podatki objekta, ki je s poljem `id` referenciran v tabeli, imenovani v `sql_field`,
- `is_disabled` – administrativna zastavica za izklop procesiranja relacije.

Primer opisanega razvoja podatkovne baze je razviden iz Vzorca kode 2.7 v Poglavju 2.1.2, Slike 3.8 in Slike 3.9, diagram končne podatkovne baze pa s Slike 3.10.

### 3.5.2 Procesna logika

Kot je razvidno s sledečih poglavij, smo pri implementaciji logike za prenos podatkov uporabili nekatere tehnike za optimizacijo delovanja, in sicer:

- predpomnjenje, s katerim smo občutno zmanjšali število branj iz podatkovne baze in število klicev nekaterih časovno potratnejših algoritmov ter posledično pohitrili delovanje logike. Več o vplivu predpomnenja na delovanje procesne logike smo zapisali v Poglavju 4.2;
- večnitenje, s katerim smo omogočili učinkovitejšo porabo procesne moči večjedrnega strežniškega procesorja in obšli čakanje med poslanimi zahtevami GET in prejetimi odzivi z Graph API-ja. Več o vplivu večnitenja na delovanje procesne logike smo zapisali v Poglavju 4.2. Razvili smo sistem za upravljanje z nitmi (angl. *thread-pool*) in med implementacijo objektov in algoritmov pazili na souporabo virov z ustreznim določanjem obsega spremenljivk (instančne, statične in nitno-statične spremenljivke), uporabo objekta `HttpContext`, zaklepanjem virov (angl. *locking*) in uporabo rezervirane besede `volatile` [24, 25];




Slika 3.8: Relacijske tabele, ki posnemajo strukturo objekta, razvidnega iz Vzorca kode 2.7 v Poglavju 2.1.2.

| sql_entity | sql_field | api_field | api_modifier | is_user_only | is_connection | is_active_reference | is_disabled |
|---|---|---|---|---|---|---|---|
| fbUser | fbUserLike | likes | .limit(150) | 0 | 1 | 1 | 0 |

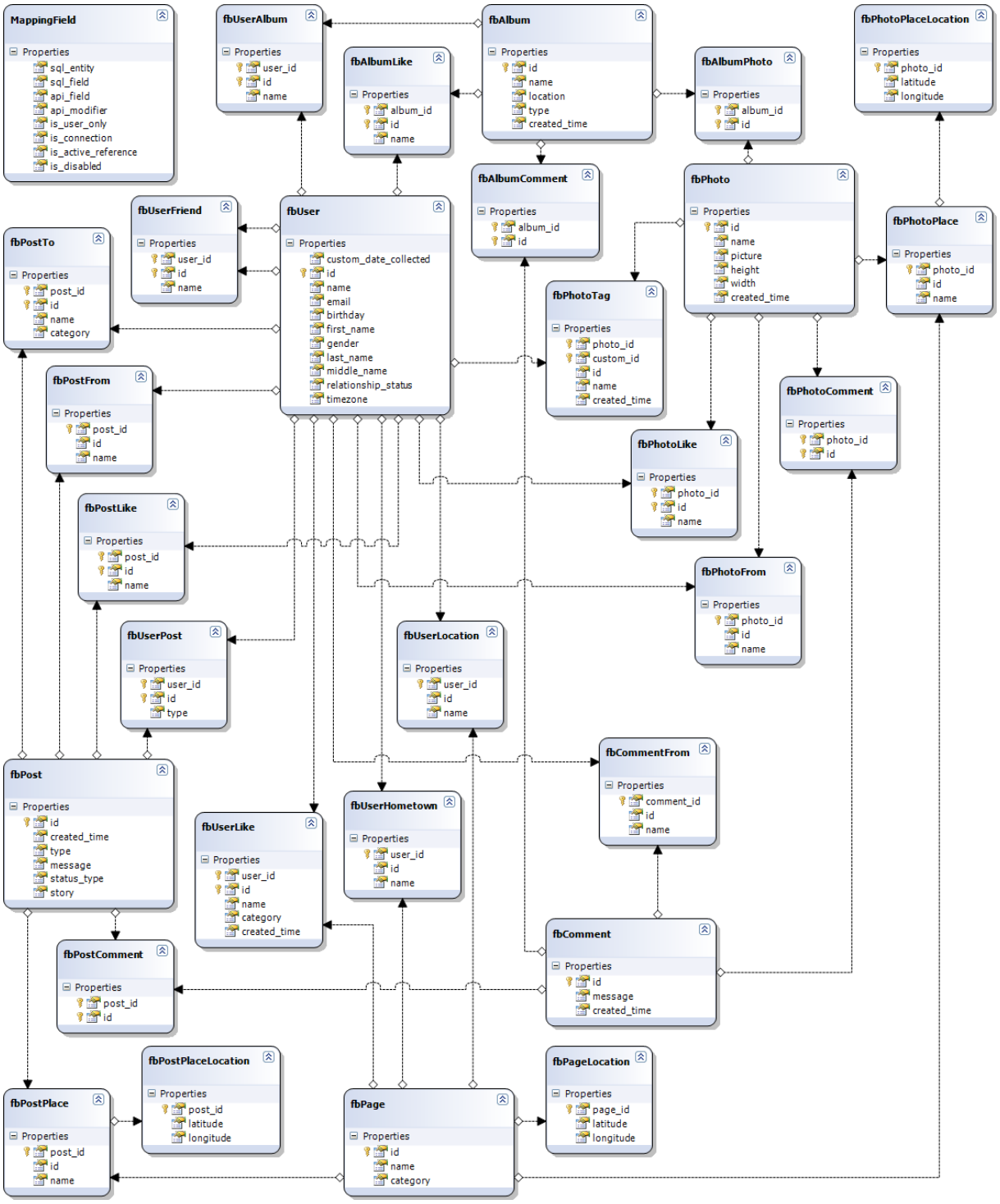
Slika 3.9: Preslikovalna tabela z vnosom za tabele in njihove medsebojne relacije, razvidne s Slike 3.8. Iz vnosa je razvidno, da gre za relacijo med tabelama `fbUser` in `fbUserLike`, ki je v Graph API-ju predstavljena s povezavo `likes`. Podatkov bo preneseno največ 150 na stran, preneseni bodo za uporabnika in njegove prijatelje, v tabelo `fbPage` pa bodo preneseni tudi podatki objektov, ki jih všečki referencirajo.

- asinhronost, s katero smo pridobili na odzivnosti uporabniškega vmesnika in omogočili določene funkcionalnosti, kot sta delovanje procesne logike ločeno od niti uporabniškega vmesnika, torej brez konstantne povezave med strežnikom in odjemalcem, ter možnost elegantnega preklica zbiranja podatkov med delovanjem z uporabo gumba za prekinitev.

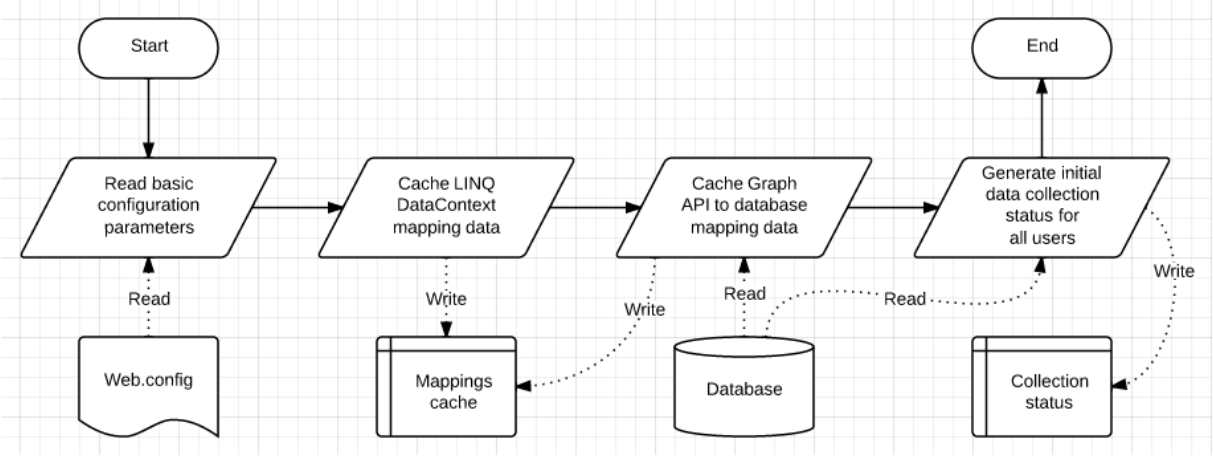
- dinamično programiranje z uporabo dinamičnih objektov, generičnih metod in refleksije, s katerim smo zmanjšali ponavljanje in količino kode ter povečali njeno obvladljivost. Čeprav je izvajanje dinamične kode, še posebej pri uporabi refleksije, v jeziku C# v večini primerov počasnejše od eksplicitne [5, 6], smo se za to odločili zaradi možnosti hitrih nadgradenj in sprememb, saj smo predpostavili, da se delovanje Graph API-ja in struktura Facebookovega socialnega grafa pogosto spreminjata. Naša predpostavka se je izkazala za ustrezno, saj je v času razvoja aplikacije *OKG* trikrat prišlo do večjih sprememb (angl. *breaking changes*) omenjenih Facebookovih komponent.

Na kritičnih predelih kode smo implementirali bloke `try-catch` za rokovanje z morebitnimi napakami med izvajanjem aplikacije [26], s čimer smo znatno izboljšali njeno stabilnost. Uporabili smo tudi sledenje (angl. *tracing*), s katerim smo med izvajanjem, še posebej pri napakah v omenjenih `try-catch` blokih, zapisovali določene informacije v datoteko na disku, ki nam je bila v pomoč pri razhroščevanju aplikacije. Implementirali smo še objekte štoparice, s katerimi smo merili časovno zahtevnost Algoritma za prenos podatkov (Poglavje 4.2).

Med implementacijo logike, ki uporablja Graph API, smo poleg že omenjene nezanesljivosti njegove dokumentacije, zasledili tudi nekonsistentnosti v njegovih odzivih, ki so se pokazale kot občasna polja brez informacij in duplikati v podatkih. Ta nekonsistentnost nam je med razvojem aplikacije *OKG* povzročila nemalo preglavic.




Slika 3.10: Diagram tabel in njihovih medsebojnih povezav v podatkovni bazi za hranitev uporabniških podatkov in delovanje logike za prenos podatkov.




Slika 3.11: Entitetno-relacijski diagram inicializacije aplikacije *OKG* na strežniku.

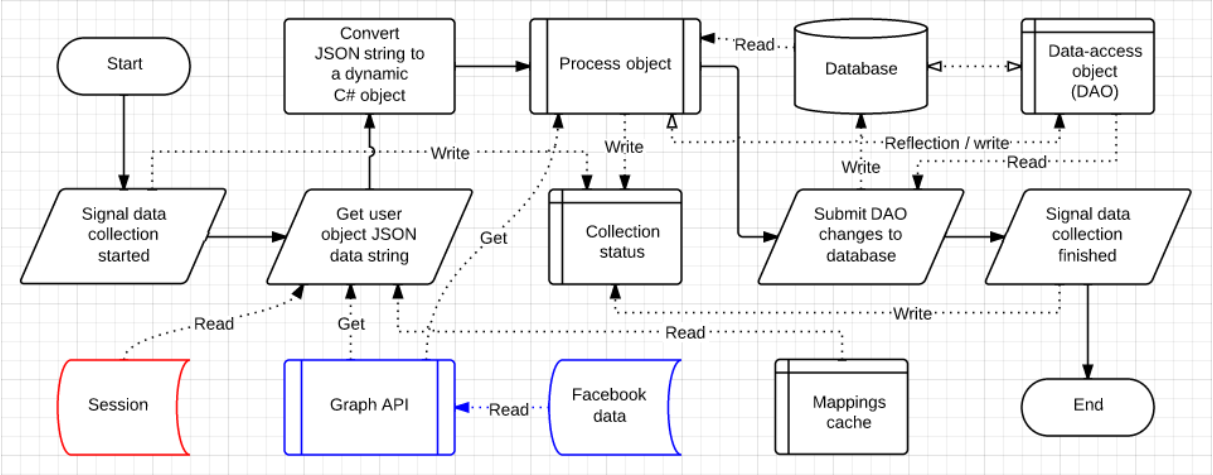
#### 3.5.2.1 Inicializacija aplikacije

Ob zagonu spletne aplikacije na strežniku se, kot je razvidno z diagrama na Sliki 3.11, izvede začetna inicializacija aplikacije: iz konfiguracijske datoteke `Web.config` se v pomnilnik prenesejo nastavitveni parametri aplikacije, vključno z URL-jem spletne strani, identifikacijsko številko in skrivnostjo aplikacije na spletni strani Facebook Developers ter parametri za delovanje večnitenja, asinhronosti, predpomnenja in sledenja. Tu gre za podatke, potrebne za osnovno delovanje aplikacije, ki se med izvajanjem ne spreminjajo.

Nato se izvede predpomnenje kompleksnejših podatkov, do katerih aplikacija pogosto dostopa med izvajanjem, spreminja pa jih načeloma ne. Tu so vključeni podatki Preslikovalne tabele, opisane v Poglavju 3.5.1, in podatki o strukturi same podatkovne baze, ki jih aplikacija razbere iz objekta `DataContext` z uporabo metod imenskega prostora `System.Data.Linq.Mapping`[2].

Ob koncu inicializacije se v pomnilniku ustvari še slovar vseh v podatkovni bazi obstoječih objektov `user` z vrednostmi, ki povedo stanje o podatkih določenega uporabnika. Slovar lahko vsebuje vrednosti za naslednja stanja podatkov: niso bili nikoli preneseni, so bili preneseni v preteklosti, se trenutno prenašajo, so se v zadnjem prenosu uspešno prenesli, se v zadnjem prenosu niso uspešno prenesli. Slovar smo v nadaljevanju imenovali *Stanje prenosa* (angl. *Collection status*).

[2] Več informacij na naslovu:
http://msdn.microsoft.com/en-us/library/system.data.linq.mapping.aspx




Slika 3.12: Entitetno-relacijski diagram delovanja Algoritma za prenos podatkov. Rdeče obarvana entiteta se nahaja na odjemalcu, modro obarvani na Facebookovem strežniku, črno obarvane pa na strežniku aplikacije *OKG*. Pikčaste črte z napisom Get predstavljajo zahteve GET.

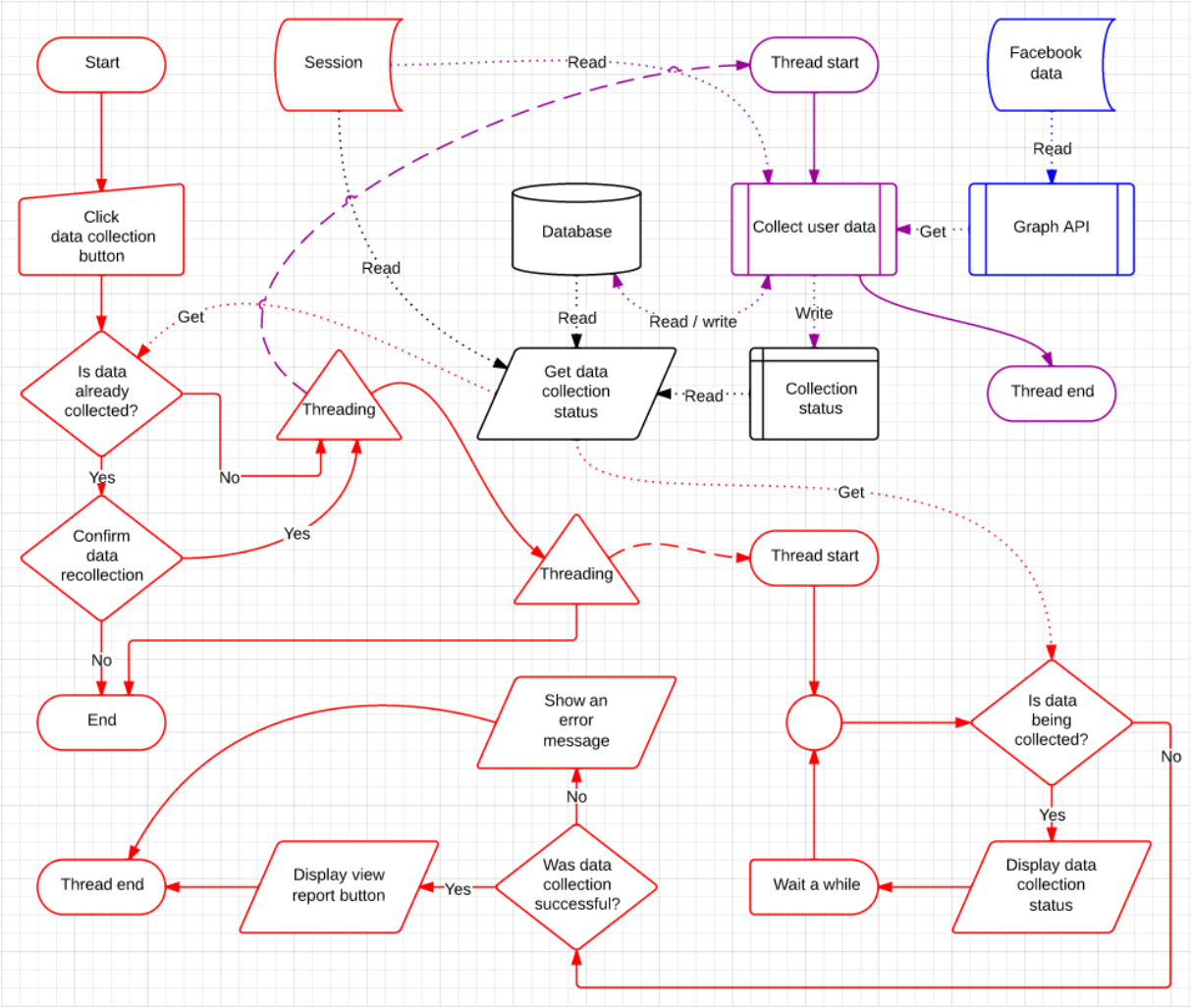
#### 3.5.2.2 Stran za prenos podatkov

Kot je razvidno z diagrama na Sliki 3.13, se ob uporabnikovem kliku na gumb za prenos podatkov na Strani za prenos podatkov (Poglavje 3.3) sproži algoritem, ki s strežnika zahteva vrednost Stanja prenosa za trenutnega uporabnika in v primeru že prenesenih podatkov prikaže dialog za potrditev ponovnega prenosa. Po potrditvi oz. v primeru še ne prenesenih podatkov omenjen algoritem požene dve niti, in sicer Nit za prenos podatkov in Nit za preverjanje vrednosti Stanja prenosa. Prva se izvaja na strežniku, druga pa na uporabniškem vmesniku.

Prva med svojim izvajanjem, opisanem v Poglavju 3.5.2.3, ustrezno zapisuje vrednosti Stanja prenosa, druga pa jih intervalno preverja. Ko druga razbere, da se je izvajanje prenosa uporabniških podatkov zaključilo, preveri uspešnost prenosa in prikaže ustrezno vsebino na uporabniškem vmesniku. Ob uspešnem prenosu se na Strani za prenos podatkov prikaže gumb za pregled analitičnega poročila, o katerem smo več zapisali v Poglavju 3.6.

#### 3.5.2.3 Algoritem za prenos podatkov

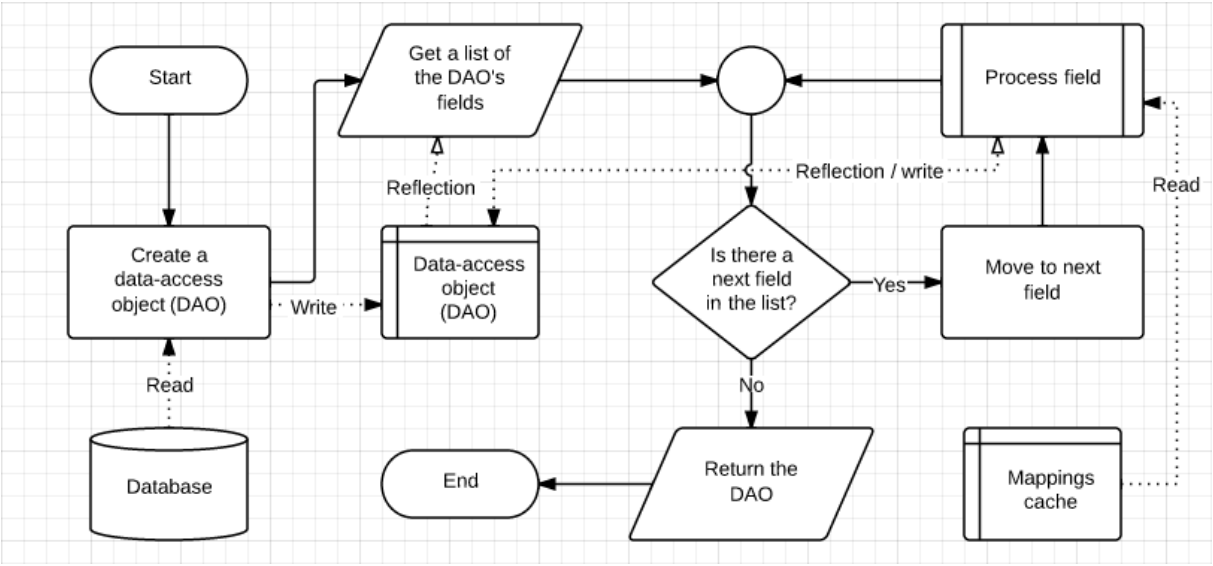
Nit za prenos podatkov izvede Algoritem za prenos podatkov, katerega delovanje je razvidno z diagrama na Sliki 3.12. Slednji na začetku v Stanju prenosa označi začetek prenosa in iz uporabniškega identitetnega objekta v seji brskalnika pridobi uporabnikov




Slika 3.13: Entitetno-relacijski diagram delovanja Strani za prenos podatkov. Rdeče obarvane entitete se izvajajo oz. nahajajo na odjemalcu, modro obarvane na Facebookovem strežniku, črno in vijolično obarvane pa na strežniku aplikacije *OKG*. Pikčaste črte z napisom Get predstavljajo zahteve GET.

dostopni žeton. Ta mu omogoči, da nato od Graph API-ja, kot je opisano v Poglavju 2.1.2, zahteva podatke o uporabniku in njegovih prijateljih. Za določitev obsega zahtevanih podatkov algoritem uporabi predpomnjene informacije o Preslikovalni tabeli in strukturi podatkovne baze. Podatke, prejete z Graph API-ja v obliki JSON, pretvori v dinamični objekt jezika C# (`dynamic`, v nadaljevanju *DO*), katerega nato obdela z Algoritmom za procesiranje DO-jev, opisanim v sledečem odstavku. Po končani obdelavi DO-ja pridobi objekt za manipulacijo in dostop do podatkovne baze (v nadaljevanju DAO), ki vsebuje obdelane uporabniške podatke. Slednje nato preko metod objekta DAO zapiše v podatkovno bazo. Na koncu izvajanja algoritem še v Stanje prenosa zapiše vrednost, ki predstavlja končan prenos podatkov.

Algoritem za procesiranje DO-jev prejme DO in preko objekta `DataContext` inicializira DAO ustreznega tipa – npr. če gre za DO, ki predstavlja Facebookov objekt `user`, inicializira DAO tipa `fbUser`. Z uporabo refleksije nato pridobi seznam podatkovnih polj (angl. *fields*) in lastnosti (angl. *properties*) objekta DAO, ki predstavljajo stolpce in relacije ustrezne relacijske tabele v podatkovni bazi (Poglavje 3.5.1). Algoritem se nato sprehodi po seznamu in obdela vsa polja in lastnosti z Algoritmom za procesiranje dinamičnih polj in lastnosti, opisanim v sledečem odstavku, ki v DAO ustrezno skopira podatke iz DO-ja. Na koncu algoritem klicatelju vrne objekt DAO, ki na tem mestu vsebuje vse podatke DO-ja v obliki, primerni za vpis v podatkovno bazo. Opisano delovanje algoritma je razvidno z diagrama na Sliki 3.14.




Slika 3.14: Entitetno-relacijski diagram algoritma za procesiranje DO-jev.

Algoritem za procesiranje dinamičnih polj in lastnosti, katerega delovanje je razvidno z diagrama na Sliki 3.15, prejme seznam polj objekta DAO in za vsako polje preveri veljavnost glede na pripadajočo zastavico `is_disabled` v Preslikovalni tabeli (Poglavje 3.5.1) in obstoj polja z enakim imenom in veljavnimi podatki v DO-ju. Če je polje veljavno, algoritem razbere kakšnega tipa je vrednost tega polja v objektu DO.

Prva možnost predstavlja vrednost primitivnega tipa – številko, znak ali niz znakov. Ta se iz DO-ja preprosto skopira v DAO. V primeru, ko vrednost predstavlja identifikacijsko številko nekega drugega objekta Facebookovega socialnega grafa in je vključena pripadajoča zastavica `is_active_reference` v Preslikovalni tabeli (Poglavje 3.5.1), se nad njo požene Nit za obdelavo referenčnega objekta, algoritem pa nadaljuje z naslednjim poljem oz. lastnostjo v seznamu.

Nit za obdelavo referenčnega objekta prejme identifikacijsko številko objekta Facebookovega socialnega grafa in izvede algoritem, ki je podoben algoritmu Niti za prenos podatkov: preko Graph API-ja pridobi podatke objekta v obliki JSON, jih pretvori v DO in le-tega obdela z Algoritmom za procesiranje DO-jev. Po obdelavi dobi ustrezni DAO in preko njega shrani podatke referenčnega objekta v podatkovno bazo.

Druga možnost predstavlja kompleksno vrednost oz. podrejen DO. Nad takšno vrednostjo se izvede Algoritem za procesiranje DO-jev, ki vrne nov podrejeni DAO, ta pa se pripne na obstoječi nadrejeni DAO. Njegovi podatki se v podatkovno bazo vpišejo skupaj s podatki nadrejenega objekta DAO ob koncu izvajanja Niti za prenos podatkov.

Tretja in zadnja možnost predstavlja seznam objektov oz. več podrejenih DO-jev. Ker Graph API pri seznamih uporablja ostranjevanje podatkov (Poglavje 2.1.2), tak seznam vsebuje le prvo stran objektov. Algoritem požene Nit za obdelavo strani, nato pa, dokler obstajajo naslednje strani podatkov, le-te postopoma pridobi preko Graph API-ja in nad vsako požene novo Nit za obdelavo strani.

Nit za obdelavo strani se sprehodi po prejetem seznamu podrejenih DO-jev, za vsakega pridobi nov podrejeni DAO z Algoritmom za procesiranje DO-jev, in ga pripne na obstoječi nadrejeni DAO.


Slika 3.15: Entitetno-relacijski diagram delovanja Algoritma za procesiranje dinamičnih polj in lastnosti. Modro obarvani entiteti se izvajata oz. nahajata na Facebookovem strežniku, črno in vijolično obarvane pa na strežniku aplikacije *OKG*. Pikčasti črti z napisom Get predstavljajata zahtevi GET.

## 3.6 Analiza uporabniških podatkov

Po prenosu svojih podatkov s Facebooka v aplikacijo *ObrazoKnjigoGled*, uporabniki pridobijo možnost ogleda Poročila – analize teh podatkov. V sledečih poglavjih smo opisali način generacije vizualizacij s knjižnico *Charting* (Poglavje 3.6.1) in implementacijo geokodiranja v knjižnici *Geocoding* (Poglavje 3.6.2). Omenjeni projektni knjižnici se uporabljata v logiki knjižnice *OKG.Reporting* pri dinamični generaciji Poročila, sproženi ob kliku uporabnika na gumb za pregled poročila na Strani za prenos podatkov (Poglavje 3.3). Vsebino poročila smo opisali v Poglavju 3.6.3.

### 3.6.1 Vizualizacije

V Poročilu smo uporabnikom določene informacije o njihovih podatkih prikazali v vizualizacijah različnih tipov. V namen generiranja vizualizacij med izvajanjem aplikacije *OKG* smo izvedli korake, opisane v nadaljevanju.

V knjižnici *Core.Xml* smo v jeziku C# definirali objekta `XmlTag` in `HtmlTag`, ki predstavljata znački XML in HTML ter se lahko vanju tudi ustrezno serializirata. Objekta vključujeta metode za določanje atributov značk – pri objektu `HtmlTag` vključno z identifikatorjem (atribut `id`), razredi (atribut `class`) in slogi (atribut `style`). Poleg omenjenih objektov smo definirali še vmesnik `IHtmlable`, ki v svojih dedujočih objektih zahteva implementacijo metode `ToHtml`, s katero trenutni objekt ustrezno spremenimo v objekt `HtmlTag`.

Implementirali smo knjižnico *Charting*, v kateri smo v jeziku C# definirali generični objekt `ChartBox`, ki deduje vmesnik `IHtmlable` in vsebuje kolekcijo objektov `Chart`. `Chart` je abstraktni objekt, katerega dedujejo vsi objekti, ki predstavljajo vizualizacije. Na objektu `ChartBox` smo omogočili nastavljanje opcij, kot so naslov vsebovane skupine vizualizacij, format pojasnjevalnega in dodatnega sporočila, ki sta v Poročilu prikazana ob premiku miškinega kazalca na ustrezni element, razrede za uporabo v slogih CSS, preklopke (angl. *toggles*) med vsebovanimi vizualizacijami in značko, s katero določimo, ali se vizualizacije izrišejo takoj po kreaciji, ali šele ob kliku uporabnika na gumb. Slednjo zastavico smo vklopili pri procesno zahtevnejših vizualizacijah, saj se algoritmi za izris vizualizacij izvajajo na uporabniškem vmesniku in uporabljajo vire odjemalca. Objekt `ChartBox` poleg metode `ToHtml` vsebuje še metodo `ToJavaScript`, s katero pridobimo kodo JS za izris in interakcijsko funkcionalnost njegovih vsebovanih vizualizacij.

Definirali smo objekta `GoogleChart` in `D3Chart`, ki dedujeta objekt `Chart`, in pred-

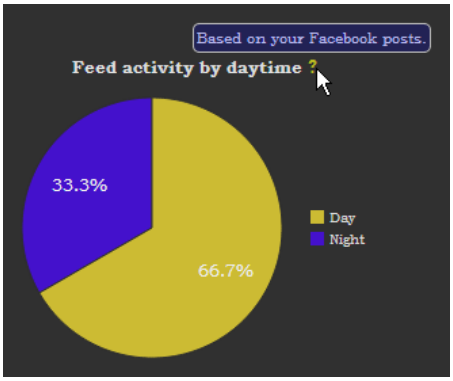
| tip vizualizacije | izvorno ime | izvor |
|---|---|---|
| krožni grafikon | Pie Chart | Google Charts |
| stolpčni diagram | Column Chart | |
| palični diagram | Bar Chart | |
| črtni grafikon | Line Chart | |
| mehurčni grafikon | Bubble Chart | |
| geografski grafikon | Geochart | |
| časovnica z označbami | Annotated Timeline | |
| zemljevid | Map | |
| mrežni graf | Force Layout | D3 |
| akordni diagram | Chord Diagram | |
| besedni oblak | Word Cloud | |

Tabela 3.2: Seznam tipov vizualizacij, podprtih v razviti knjižnici *Charting*, in njihovih izvornih imen v API-ju Google Charts oz. knjižnici D3.

stavljata generične vizualizacije – pri prvem gre za preprostejše vizualizacije API-ja Google Charts, pri drugem pa za kompleksnejše vizualizacije knjižnice D3. Nato smo izdelali objekte v obliki ovojnic (angl. *wrapper*) kode JS, ki se uporablja za prikaz določenega tipa vizualizacij na uporabniškem vmesniku. Omenjeni objekti izpostavljajo metode in lastnosti, preko katerih lahko v tej kodi urejamo nastavitve želene vizualizacije. Nastavitveni parametri vizualizacij so opisani v spletni dokumentaciji API-ja Google Charts in knjižnice D3. Seznam vizualizacij, ki smo jih raziskali in implementirali v knjižnici *Charting*, je naveden v Tabeli 3.2.

Pri uporabi opisanih objektov v namen generiranja vizualizacij je treba izvesti sledeče korake (ne nujno v navedenem vrstnem redu):

- inicializacija in konfiguracija objekta `ChartBox`,
- inicializacija in konfiguracija posameznih vizualizacij z objekti, ki dedujejo `Chart`, in njihova vključitev v `ChartBox`,
- izračun ustreznih podatkov za prikaz v vizualizacijah,
- ustrezna določitev izračunanih podatkov objektom, ki predstavljajo vizualizacije v objektu `ChartBox`, s klici metod `SetData`,
- klic metod `ToHtml` in `ToJavaScript` na objektu `ChartBox` za serializacijo vizualizacij v HTML in JS.




Slika 3.16: Krožni grafikon, zgeneriran s kodo v Vzorcu kode 3.1.

Primer te uporabe je razviden iz Vzorca kode 3.1. Za podrobnejše razumevanje delovanja objektov in njihovih metod v navedenem primeru smo v dodatku navedli še izvorno kodo vmesnika `IHtmlable`, objektov `HtmlTag`, `ChartBox`, `Chart` in `GoogleChart` ter objekta `PieChart` s pripadajočo kodo JS. Za pravilen grafični prikaz v HTML serializiranega objekta `HtmlTag`, kreiranega z metodo `ToHtml` na objektu `ChartBox`, smo morali definirati še ustrezne sloge CSS za razrede `chartbox`, `chart`, `title`, `toggle`, `infotext` in `extratext`, ter metode JS za funkcionalnost preklopk in gumba za izris vizualizacije. Kodi HTML in JS, zgenerirani v omenjenem primeru, sta razvidni iz vzorcev kode 3.2 in 3.3, grafični prikaz končnega krožnega grafikona pa s Slike 3.16.

### 3.6.2 Geokodiranje

Pri implementaciji logike za generacijo vizualizacij Poročila, ki prikazujejo lokacijske podatke, smo potrebovali funkcijo za preslikavo koordinat GPS v fizični naslov oz. t.i. obratno geokodiranje (angl. *reverse geocoding*) [27]. Implementirali smo knjižnico *Geocoding*, v kateri smo definirali objekt z informacijo o zemljepisni širini in dolžini ter naslovom fizične lokacije na Zemlji (v nadaljevanju *geolokacija*) in funkcionalnostjo obratnega geokodiranja.

Obratno geokodiranje smo prvotno implementirali z uporabo spletnega API-ja Google Maps Geocoding, ki deluje na principu zahtev GET, katerim določimo parameter z zemljepisno dolžino in širino [27]. Primer takšne zahteve smo navedli v Vzorcu kode 3.4, odziv nanjo pa v Vzorcu kode 14 v dodatku. Pri obratnem geokodiranju večjega števila geolokacij, npr. lokacij vseh slik uporabnika, pa smo naleteli na težavo, saj Google Maps Geocoding za nekomercialno rabo podpira le 2500 zahtev na dan [27]. Podobno omejitev

```
// ChartBox creation.
var piePostDayNight = new ChartBox<PieChart>();
piePostDayNight.Style.Title = "Feed activity by daytime";
piePostDayNight.Style.InfoText = "Based on your Facebook posts.";

// Data calculation.
var chartData = new Dictionary<string, int>();
foreach (fbPost post in _posts) //Loop through user's posts.
{
    var createdTime = DataUtils.GetDateTime(post.created_time);
    var location = DataUtils.GetPostLocation(post) ?? ↩
       DataUtils.GetUserLocation(_user);
    if (createdTime != null && location != null)
    {
        var key = GeoUtils.IsDaytime(location.Latitude, location.Longitude, ↩
           createdTime) ? "Day" : "Night";
        if (!chartData.Contains(key))
            chartData[key] = 1;
        else
            chartData[key] += 1;
    }
}
chartData = chartData
    .OrderByDescending(x => x.Key)
    .ToDictionary(x => x.Key, x => x.Value);

// Chart creation.
piePostDayNight.Add(new PieChart("examplepiechart", new[] { "Time of day", ↩
    "No. of Posts" }));
piePostDayNight[0].Options.Colors = new GColors("#ccbb33", "#4411cc");
piePostDayNight[0].SetData(chartData);
```

Vzorec kode 3.1: C# – primer uporabe objektov, implementiranih v knjižnici *Charting*, za izdelavo krožnega grafikona uporabnikovih objav na Facebooku glede na čas objave – podnevi ali ponoči. Metoda `GeoUtils.IsDayTime` je opisana v Poglavju 3.6.2. Ostali nestandardni objekti in metode so razvidni iz vzorcev kode v dodatku.

```
<div class="chartbox">
  <div class="title">Feed activity by daytime
    <a class="infotext">?
      <div>Based on your Facebook posts.</div>
    </a>
  </div>
  <div class="chart" id="examplepiechart"></div>
</div>
```

Vzorec kode 3.2: HTML – koda za prikaz krožnega grafikona na Sliki 3.16, generirana ob klicu metode `ToString` na objektu, generiranem ob klicu metode `ToHtml` na objektu `piePostDayNight`, kreiranem v Vzorcu kode 3.1.

```
var examplepiechart = { chart: null };

var examplepiechartOptions = {
    colors: ['#ccbb33', '#4411cc'],
    chartArea: {left: 10, top: 0, width: '100.000%', height: '100.000%'},
    fontSize: 13,
    ⋮
};

function Drawexamplepiechart() {
    var data = google.visualization.arrayToDataTable([
        ['Daytime','Items'],
        ['Day', 424],
        ['Night', 212]
    ]);
    if (!examplepiechart.chart) {
        examplepiechart.chart = new google.visualization.PieChart( ↩
            document.getElementById('examplepiechart'));
        examplepiechart.chart.draw(data, examplepiechartOptions);
    }
}
Drawexamplepiechart();
```

Vzorec kode 3.3: JS – koda za prikaz krožnega grafikona na Sliki 3.16, generirana ob klicu metode `ToJavaScript` na objektu `piePostDayNight`, kreiranem v Vzorcu kode 3.1.

```
GET http://maps.googleapis.com/maps/api/geocode/json?
    latlng=46.044843,14.489293
   &sensor=false
```

Vzorec kode 3.4: Primer zahteve GET za obratno geokodiranje koordinat GPS preko API-ja Google Maps Geocoding. Odziv nanjo smo navedli v Vzorcu kode 14 v dodatku.

smo zasledili tudi pri drugih spletnih API-jih za geokodiranje, kot sta npr. *Bing Maps REST Services Locations*[3] in *Yahoo! BOSS Geo Services PlaceFinder*[4].

Za rešitev omenjenege problema smo se odločili geokodiranje implementirati lokalno z uporabo podatkovne baze in podatkov geografske podatkovne baze *GeoNames*[5]. Slabo stran te implementacije pa je predstavljala velika poraba virov strežnika in posledično velika časovna zahtevnost delovanja, saj podatki vsebujejo informacije nad 10 milijonov geolokacij. Za nekoliko hitrejše delovanje smo iz podatkov izločili in uporabili približno 2,8 milijona poseljenih lokacij, vendar pa ta rešitev še vedno ni bila primerna procesiranju večjega števila zahtev; prišla je v poštev pri majhnem številu zahtev v primeru dosežene limite na spletnem API-ju.

Problema limite spletnih API-jev in časovne potratnosti lokalne implementacije smo obšli z žrtvovanjem natančnosti geokodiranja. V podatkovno bazo smo za vsako državo vnesli njene mejne zemljepisne dolžine in širine ter ob zahtevi po obratnem geokodiranju geolokacije vrnili le ime ustrezne, geolokaciji najbližje, države.

Poleg objekta z opisano možnostjo obratnega geokodiranja smo v knjižnici *Geocoding* implementirali še metode za računanje s podatki geolokacij, vključno z metodami za:

- izračun časovnega odmika geolokacije oz. določene zemljepisne dolžine od UTC-ja (prva metoda v Vzorcu kode 3.5),
- izračun dolžine dneva geolokacije oz. določene zemljepisne širine za določen dan v letu z uporabo modela *CBM* (druga metoda v Vzorcu kode 3.5) [28],
- določanje dneva in noči geolokacije ob določenem času in datumu (zadnja metoda v Vzorcu kode 3.5),
- določanje kode *ISO 3166-1 alpha 2*[6] države glede na njeno ime z uporabo imenskega prostora `System.Globalization`, ali izbrane geolokacije z uporabo obratnega geokodiranja,
- določanje vrednosti parametra `region` v Googlovem Geochartu iz seznama kod ISO 3166-1 alpha 2 držav,

[3] Dokumentacija dostopna na naslovu: http://msdn.microsoft.com/en-us/library/ff701715.aspx
[4] Dokumentacija dostopna na naslovu: http://developer.yahoo.com/boss/geo
[5] Dostopno na naslovu: http://www.geonames.org
[6] Več informacij na naslovu: http://en.wikipedia.org/wiki/ISO_3166-1_alpha-2

- računanje razdalj med geolokacijami z uporabo haversinske formule [29] in API-jem Google Distance Matrix,
- grupiranje geolokacij glede na medsebojno razdaljo.

```
// Gets the time offset from UTC in [hours] for a given longitude.
public static double GetTimeOffset(decimal longitude)
{
    return (double)(longitude / 15);
}

// Gets the day length in [hours] for a specific latitude and day of year (CBM model).
public static double GetDayLength(decimal latitude, int dayOfYear, double daylengthcoefficient = 0.833)
{
    double earthRevolutionAngle = 0.2163108 + 2.0 * Math.Atan(0.9671396 * Math.Tan(0.00860 * (dayOfYear - 186.0)));
    double sunDeclinationAngle = Math.Asin(0.39795 * Math.Cos(earthRevolutionAngle));
    double inner = Math.Sin(daylengthcoefficient * Math.PI / 180) + Math.Sin((double)latitude * Math.PI / 180) * Math.Sin(sunDeclinationAngle); // should be within [-1, 1]
    return 24.0 - (24.0 / Math.PI) * Math.Acos(inner);
}

// Gets whether a point on Earth is having daytime or nighttime at a specified UTC date and time.
public static bool IsDaytime(decimal latitude, decimal longitude, DateTime dateTime)
{
    DateTime now = dateTime.AddHours(GetTimeOffset(longitude));
    double dayLength = GetDayLength(latitude, now.DayOfYear);
    double currentHour = now.Hour + (now.Minute / 60.0) + (now.Second / 3600.0) + (now.Millisecond / 360000.0);
    return (12 - dayLength / 2) < currentHour && currentHour < (12 + dayLength / 2);
}
```

Vzorec kode 3.5: Metode za izračune, povezane z lokalnim časom geolokacij.

### 3.6.3 Poročilo

Poročilo o analizi uporabniških podatkov smo razdelili v pet vsebinskih sekcij, katerih vsebino smo opisali v nadaljevanju. Sekcija o osnovnih in lokacijskih podatkih uporabnika (Slika 1 v dodatku) vključuje:

- identifikacijsko številko uporabnika v Facebookovem socialnem grafu ter njegovo polno ime, spol, časovni pas in rojstni dan,
- izračunano starost uporabnika v dnevih in urah, čas do njegovega naslednjega rojstnega dne in njegovo nebesno znamenje,
- koordinate GPS uporabnikovega domačega mesta in trenutne lokacije ter mesto in državo naslova, pridobljenega z obratnim geokodiranjem njunih koordinat,
- zračno razdaljo med omenjenima lokacijama, izračunano z uporabo haversinske formule [29], in čas, ki je potreben za prelet te razdalje s hitrostjo 885 km/h – tipično potovalno hitrostjo modernega potniškega letala [30],
- razdalja in čas potovanja med omenjenima lokacijama z avtom in peš; vrednosti so izračunane z uporabo API-ja Google Maps Distance Matrix,
- zemljevid, s katerega je razvidna zračna razdalja med omenjenima lokacijama.

Sekcija o albumih, slikah in označbah (sliki 2 in 3 v dodatku) vključuje:

- število naloženih albumov in slik, število označb (angl. *tags*) na teh slikah in število vseh slik, v katerih je označen uporabnik,
- povprečje, standardno deviacijo, minimum, maksimum in mediano (v nadaljevanju *psdmmm.*) števila slik na album, časovnega razpona slik v albumih, števila všečkov in komentarjev na album in posamezno sliko ter števila označb na sliko,
- povprečno velikost naloženih slik glede na njihovo število pik in povprečno razmerje prikaza (angl. *aspect ratio*),
- krožne grafikone albumov glede na število všečkov, komentarjev in všečkov na sliko, slik glede na tip in orientacijo ter ljudi glede na število označb v uporabnikovih slikah, na število označb z uporabnikom v isti sliki in glede na število naloženih fotografij, v katerih je uporabnik označen,

- mehurčni grafikon, ki prikazuje korelacijo med številom všečkov in komentarjev naloženih slik,

- zemljevid, ki prikazuje lokacije naloženih slik s točkami, katerih velikost je odvisna od števila slik na posamezni lokaciji,

- časovnico, ki prikazuje število naloženih slik in albumov ter število označb uporabnika po času s posebnimi opombami ob dnevih z velikimi spremembami v določenem številu.

Sekcija o prijateljih (slike 4, 5 in 6 v dodatku) vključuje:

- število prijateljev uporabnika, psdmmm. števila skupnih prijateljev z uporabnikom na prijatelja in palična diagrama imen prijateljev z največ skupnimi prijatelji ter števila prijateljev po številu skupnih prijateljev,

- psdmmm. starosti prijateljev, število in odstotek rojstnih dni prijateljev v naslednjem tednu in mesecu ter palična diagrama najstarejših in najmlajših prijateljev,

- krožni grafikon spolov prijateljev, stolpčni diagram vrednosti oz. deležev prijateljev po starosti in spolu ter rojstnem mesecu in spolu, palični diagram vrednosti oz. deležev prijateljev po nebesnem znamenju in spolu,

- število unikatnih imen, drugih imen in priimkov prijateljev, število prijateljev z dvema imenoma in palična diagrama najpogostejših imen in priimkov glede na spol,

- število in delež samskih prijateljev, krožni grafikon zakonskih stanov za vse, moške in ženske prijatelje, palični diagram distribucije vrednosti oz. deležev zakonskih stanov po starosti prijateljev ter akordni diagram, ki prikazuje nebesna znamenja, povezana glede na prijatelje v medsebojnih zvezah,

- psdmmm. razdalj med domačimi kraji in trenutnimi lokacijami prijateljev ter lokacijo uporabnika in trenutnimi lokacijami prijateljev, prijatelje, ki so najbolj oddaljeni od uporabnika, zemljevida domačih krajev in trenutnih lokacij prijateljev, krožni grafikon porazdelitve dneva in noči glede na trenutne lokacije prijateljev, stolpčni diagram z distribucijo prijateljev po časovnem odmiku od UTC-ja in črtni grafikon porazdelitve prijateljev po logaritmu razdalje med njihovo in uporabnikovo trenutno lokacijo.

Sekcija o mreži prijateljev (Slika 7 v dodatku) vsebuje:

- mrežni graf, katerega točke predstavljajo uporabnikove prijatelje, njihove medsebojne povezave pa povedo, ali sta dva prijatelja tudi med seboj prijatelja. Velikost točk je odvisna od števila povezav posamezne točke, barve točk pa glede na uporabnikovo izbiro predstavljajo spol, zakonski stan ali starost uporabnikov. Graf je mogoče prikazati za moške, ženske in vse prijatelje.

Sekcija o aktivnosti na Facebooku (Slika 8 v dodatku) vsebuje:

- število vseh uporabnikovih objav,
- dan in uro v dnevu, ko je uporabnik objavil največ objav, s številom teh objav in njihovim deležem glede na vse objave,
- krožni grafikon aktivnosti uporabnika glede na dan in noč in mehurčni grafikon, ki prikazuje korelacijo števila objav med dnevom in uro v dnevu,
- besedni oblak izrazov v uporabnikovih objavah.

# Poglavje 4

# Uporaba aplikacije

V sledečih poglavjih smo zapisali informacije o uporabi aplikacije *ObrazoKnjigoGled* v praksi. Zaradi časovne omejitve diplomskega dela aplikacije sicer nismo uporabili v produkcijskem večuporabniškem okolju, smo jo pa večkrat testirali s svojim uporabniškim profilom na razvojnem računalniku. V Poglavju 4.1 smo zapisali navodila za uporabo aplikacije, kar smo v Poglavju 4.2 nadaljevali z opisom in rezultati meritev časovne zahtevnosti Algoritma za prenos podatkov glede na izbrane konfiguracijske parametre.

## 4.1 Navodila za uporabo

Uporabnik do spletne aplikacije *ObrazoKnjigoGled* dostopa z ustreznim URL-jem[1] preko spletnega brskalnika. Ob prvem dostopu se mu prikaže Stran za vpis (Slika 4.1), ki vključuje gumb za vpis v aplikacijo preko Facebooka. Ob kliku na omenjeni gumb se prikaže Facebookov dialog za avtentikacijo, preko katerega se uporabnik z uporabniškim imenom in geslom vpiše v svoj Facebook profil in aplikaciji *OKG* odobri osnovna dovoljenja za uporabo svojih podatkov (Slika 2.2 v Poglavju 2.1.1).

Med izvajanjem aplikacije se v zgornjem desnem kotu izpisujejo razna sporočila, ki jih lahko uporabnik skrije s klikom. Sporočila na rdečem ozadju predstavljajo napake, na modrem splošne informacije, na zelenem pa informacije o uspešno izvedenih procesih.

V splošnem rumeni znaki in besede na spletni strani predstavljajo elemente, na katerih se ob kliku ali premiku miškinega kazalca zgodi določen dogodek.

Po uspešni avtentikaciji se uporabniku prikaže Stran za prenos podatkov (Slika 4.2), ki vključuje gumb za generacijo Poročila oz. prenos uporabniških podatkov s Facebooka

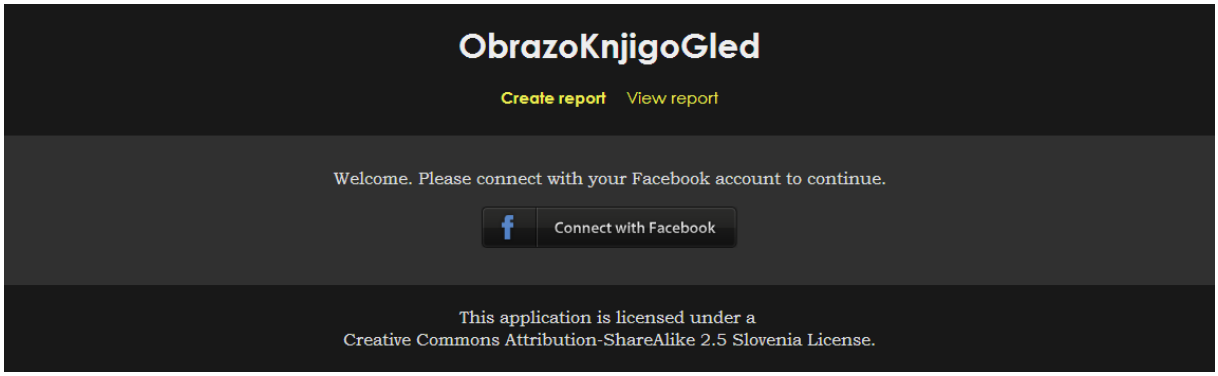
[1] V času pisanja diplomske naloge še neobstoječ.

in zavihka za odobritev dodatnih dovoljenj aplikaciji in izpis iz aplikacije. Ob pomiku miškinega kazalca na posamezno dovoljenje se prikažejo dodatne informacije, ob kliku pa se odpre dialog za odobritev dovoljenja. Klik na gumb za generacijo poročila sproži Algoritem za prenos podatkov, ki lahko traja od nekaj minut do ene ure – odvisno od izbranih dovoljenj, količine podatkov na uporabnikovem profilu in zasedenosti Facebookovih strežnikov. Med prenosom podatkov lahko uporabnik spletno stran zapre in se vrne kasneje; ob naslednjem dostopu do aplikacije bo obveščen o stanju podatkov. Če želi prekiniti prenos podatkov, lahko to stori s klikom na gumb za preklic (Slika 4.3).

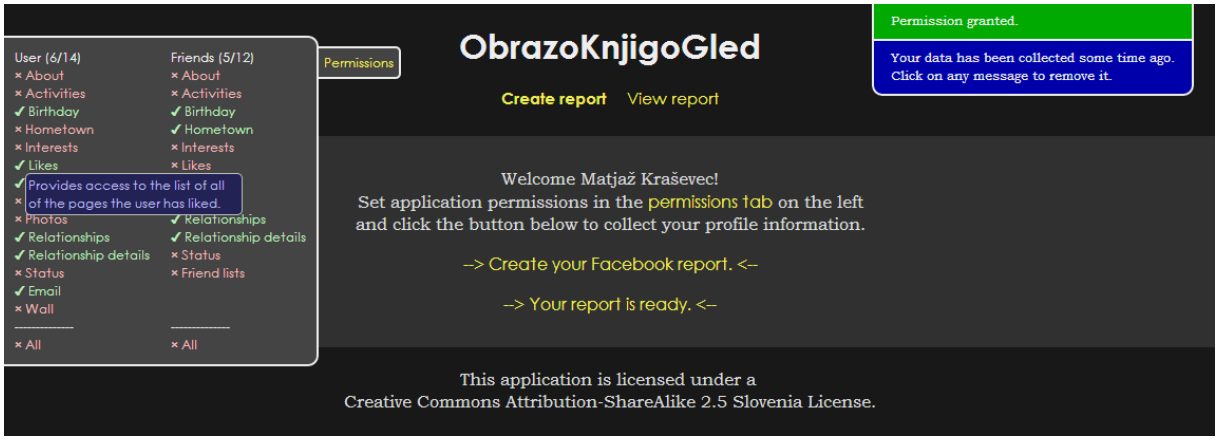
Po uspešnem prenosu podatkov se uporabniku na Strani za prenos podatkov prikaže gumb za ogled Poročila (Poglavje 3.6.3). Ob kliku na gumb se po krajšem nalaganju prikaže prva sekcija Poročila, uporabnik pa se lahko med sekcijami pomika s tipkama za levo in desno ali s kliki na navigacijske gumbe v vogalih poročila. Poročilo vsebuje elemente, občutljive na premik kazalca, in sicer:

- rumene vprašaje, ki prikažejo pojasnjevalna sporočila, npr. koliko podatkov je bilo vključeno v določen izračun oz. graf, ali pa v kateri enoti je rezultat,
- rumene zvezdice, ki prikažejo dodatne oz. podrobnejše informacije, npr. podrobnejši seznam deležev, katerih del je prikazan v krožnem grafikonu, in
- vizualizacije, katerih interaktivnost je odvisna od tipa vizualizacije. Te vsebujejo tudi elemente, občutljive na klik, s katerimi lahko uporabnik med drugim tudi preklaplja med načini prikaza vizualizacije in/ali določi obseg podatkov, uporabljenih v vizualizaciji. Za izris nekaterih procesno zahtevnejših vizualizacij je potreben klik na gumb za prikaz vizualizacije.

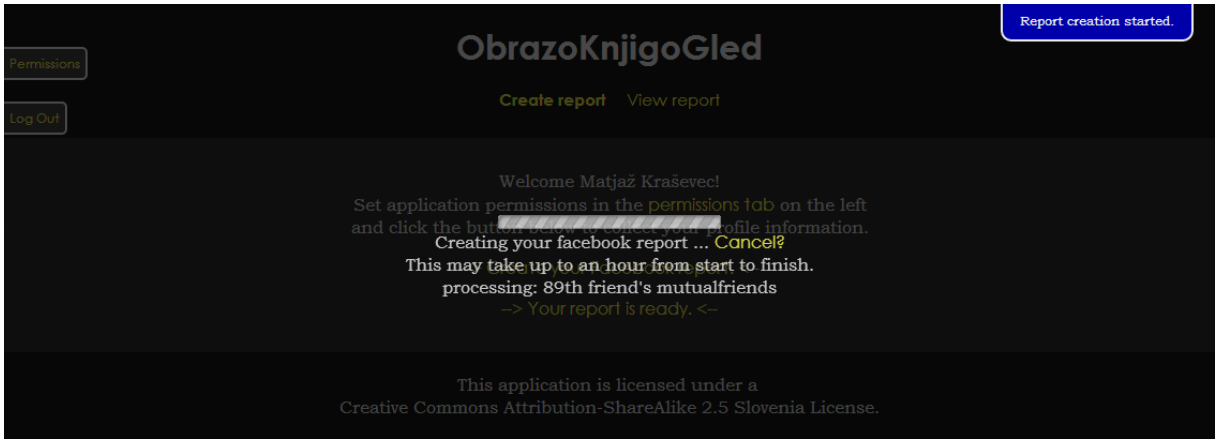
Po končani uporabi aplikacije se lahko uporabnik na strani za prenos podatkov izpiše s klikom na gumb v zavihku za izpis.




Slika 4.1: Stran za vpis.




Slika 4.2: Stran za prenos podatkov.




Slika 4.3: Aplikacija *OKG* med prenosom uporabniških podatkov s Facebooka.

## 4.2 Hitrost prenosa podatkov

Kot smo omenili v Poglavju 3.5.2, smo na določenih delih Algoritma za prenos podatkov (v nadaljevanju algoritem) implementirali objekte štoparice za meritev njegove časovne zahtevnosti. V tem poglavju smo predstavili rezultate meritev pri izvajanju algoritma glede na štiri različne konfiguracije, ki jih določata parametra:

- uporaba predpomnenja – pri izvajanju brez predpomnenja mora algoritem ob vsaki zahtevi po podatkih Preslikovalne tabele, le-te prebrati neposredno iz podatkovne baze, ob zahtevi po podatkih o strukturi podatkovne baze pa v objektu `DataContext` ustrezno najti zahtevan podatek;

- uporaba večnitenja – kot je opisano v Poglavju 3.5.2.3, se večnitenje v algoritmu uporablja za vzporedno procesiranje referenčnih objektov (Nit za obdelavo referenčnega objekta) in strani seznamov pri ostranjevanju (Nit za obdelavo strani). V primeru konfiguracije brez večnitenja se algoritem izvede linearno na eni sami niti.

Ostale konfiguracijske parametre smo pri opazovanju algoritma ohranjali konstantne. Omembe vredni so:

- obseg prenesenih podatkov, določen z vrednostmi v Preslikovalni tabeli (Poglavje 3.5.1). Izbrane vrednosti so razvidne iz Preslikovalne tabele na Sliki 4.4;

- največji možni števili vzporedno izvajajočih, trenutni niti podrejenih Niti za obdelavo strani in Niti za obdelavo referenčnega objekta. Parametra sta veljavna le v primeru večnitenja. Njuni vrednosti sta navedeni v Tabeli 4.1;

- časa čakanja določene niti pred izdelavo novih podrejenih niti, omenjenih v prejšnji točki, v primeru doseženega največjega števila vzporedno izvajajočih podrejenih niti ustreznega tipa. Tu gre za čas, po izteku katerega določena nit ponovno preveri, ali lahko kreira novo podrejeno nit. Brez teh parametrov, bi niti konstantno preverjale razpoložljivost izvajanja in s tem preprečile učinkovito izvajanje algoritma. Parametra sta veljavna le pri uporabi večnitenja. Njuni vrednosti sta navedeni v Tabeli 4.1;

- čas čakanja določene niti na obdelavo polja s seznamom objektov. Tu gre za čas, po izteku katerega določena nit ponovno preveri, ali so se vse njene podrejene Niti za obdelavo strani, ki so vezane na isti seznam objektov, zaključile, oz. če lahko nit

| sql_entity | sql_field | api_field | api_modifier | is_user_only | is_connection | is_active_reference |
|---|---|---|---|---|---|---|
| fbAlbum | fbAlbumPhoto | photos | .limit(25) | 1 | 1 | 1 |
| fbAlbum | fbAlbumLike | likes | NULL | 0 | 1 | 0 |
| fbAlbum | fbAlbumComment | comments | NULL | 0 | 1 | 1 |
| fbComment | fbCommentFrom | from | NULL | 0 | 0 | 0 |
| fbPage | fbPageLocation | location | NULL | 0 | 0 | NULL |
| fbPhoto | fbPhotoTag | tags | NULL | 0 | 1 | 0 |
| fbPhoto | fbPhotoPlace | place | NULL | 0 | 0 | 0 |
| fbPhoto | fbPhotoFrom | from | NULL | 0 | 0 | 0 |
| fbPhoto | fbPhotoLike | likes | NULL | 0 | 1 | 0 |
| fbPhoto | fbPhotoComment | comments | NULL | 0 | 1 | 1 |
| fbPhotoPlace | fbPhotoPlaceLocation | location | NULL | 0 | 0 | NULL |
| fbPost | fbPostTo | to | NULL | 0 | 1 | 0 |
| fbPost | fbPostFrom | from | NULL | 0 | 0 | 0 |
| fbPost | fbPostPlace | place | NULL | 0 | 0 | 0 |
| fbPost | fbPostLike | likes | NULL | 0 | 1 | 0 |
| fbPost | fbPostComment | comments | NULL | 0 | 1 | 1 |
| fbPostPlace | fbPostPlaceLocation | location | NULL | 0 | 0 | NULL |
| fbUser | fbUserTaggedphoto | photos | .limit(25).type(tagged) | 1 | 1 | 1 |
| fbUser | fbUserPost | feed | .limit(25) | 1 | 1 | 1 |
| fbUser | fbUserHometown | hometown | NULL | 0 | 0 | 1 |
| fbUser | fbUserLocation | location | NULL | 0 | 0 | 1 |
| fbUser | fbUserFriend | friends | .limit(3) | 0 | 1 | 1 |
| fbUser | fbUserSignificantOther | significant_other | NULL | 0 | 0 | 0 |
| fbUser | fbUserLike | likes | .limit(150) | 0 | 1 | 0 |
| fbUser | fbUserAlbum | albums | .limit(5) | 0 | 1 | 1 |
| special | special | mutualfriends | .limit(50) | 0 | 1 | 0 |

Slika 4.4: Preslikovalna tabela s privzetimi vrednostmi za aplikacijo *OKG*. S slike smo odstranili polje `is_disabled`, saj ima pri vseh vnosih vrednost `0`.

| parameter | vrednost |
|---|---|
| največje št. podrejenih Niti za obdelavo strani | 3 |
| največje št. podrejenih Niti za obdelavo referenčnega objekta | 3 |
| čas čakanja pred kreacijo Niti za obdelavo strani | 1250 ms |
| čas čakanja pred kreacijo Niti za obdelavo referenčnega objekta | 1250 ms |
| čas čakanja na obdelavo polja s seznamom objektov | 1000 ms |

Tabela 4.1: Parametri večnitenja.

nadaljuje svoje izvajanje. Parameter je veljaven le pri uporabi večnitenja. Njegova vrednost je navedena v Tabeli 4.1;

- limite največjega vrnjenega števila določenih objektov z Graph API-ja, definirane v stolpcu `api_modifier` v Preslikovalni tabeli (Slika 4.4). Limite pridejo v poštev pri ostranjevanju podatkov. Opazili smo, da v primeru prenosa podatkov brez limit, Graph API pri zahtevah po večjem številu podatkov vrne napako ali neustrezne podatke.

Algoritem smo opazovali med prenosom podatkov s svojega osebnega profila na Facebooku in pred vsakim ponovnim zagonom iz podatkovne baze pobrisali vse podatke. Razporeditev 21.424 KB podatkov, prenesenih v podatkovno bazo pri posamezni izvedbi algoritma, po relacijskih tabelah je razvidna iz Tabele 4.2. Število kreiranih niti med posamezno izvedbo algoritma z uporabo večnitenja smo navedli v Tabeli 4.4. Med delovanjem algoritma smo opravili več različnih meritev, ki so navedene v Tabeli 4.3. Za vsako meritev smo zapisali število ponovitev med posamezno izvedbo algoritma in povprečje, standardno deviacijo, minimum, mediano ter maksimum posameznih izmerjenih časov v sekundah. Rezultate meritev SUM smo navedli v Tabeli 4.5, rezultate ostalih meritev pa v Tabeli 4.6. Zaradi časovne omejitve diplomskega dela smo za posamezno konfiguracijo algoritem pognali le enkrat, torej naši rezultati niso popolnoma natančni, predstavljajo pa sprejemljiv približek. V nadaljevanju smo zapisali svoje ugotovitve o delovanju algoritma pri izbranih konfiguracijah.

Algoritem je pri večnitnem izvajanju opravil več procesiranja kot pri linearnem. Do te ugotovitve smo prišli z izračunom skupnega efektivnega časa izvajanja niti algoritma (Tabela 4.5). Slednji je pri linearnem izvajanju enak izmerjenemu celotnemu času izvajanja algoritma (meritev SUM), pri večnitnem pa smo ga izračunali iz meritev TPG, TPGW, TRF, TRFG, PAGW in SUM. Glede na posamezno nit gre tu za čas, ko se nit efektivno izvaja ali čaka na odzive Graph API-ja. Ta čas pa ne vključuje čakanja pred kreacijo podrejenih niti in čakanja na obdelavo polj s seznami podatkov, saj se slednje izvedejo v podrejenih Nitih za obdelavo strani. Pri izvajanju s predpomnenjem se je efektivni čas izvajanja algoritma z uporabo večnitenja povečal za 50%, pri izvajanju brez predpomnenja pa za kar 209%. Pri slednjem smo med izvajanjem opazili skoraj popolno zasedenost procesnih jeder, iz česar sklepamo, da je količina procesiranja v algoritmu sorazmerno odvisna od opaženega povečanja v efektivnem izvajalnem času oz. od razlike med celotnim in efektivnim izvajalnim časom.

| ime tabele | št. vnosov | količina podatkov (KB) |
|---|---|---|
| fbAlbums | 14 | 8 |
| fbAlbumComments | 12 | 8 |
| fbAlbumLikes | 63 | 8 |
| fbAlbumPhotos | 617 | 72 |
| fbComments | 1044 | 296 |
| fbCommentFroms | 1044 | 184 |
| fbPages | 113 | 32 |
| fbPageLocations | 113 | 8 |
| fbPhotos | 659 | 776 |
| fbPhotoComments | 453 | 88 |
| fbPhotoFroms | 659 | 104 |
| fbPhotoLikes | 863 | 128 |
| fbPhotoPlaceLocation | 591 | 80 |
| fbPhotoPlaces | 591 | 96 |
| fbPhotoTags | 340 | 72 |
| fbPosts | 3145 | 1296 |
| fbPostComments | 888 | 184 |
| fbPostFroms | 3145 | 608 |
| fbPostLikes | 1964 | 368 |
| fbPostPlaceLocations | 185 | 32 |
| fbPostPlaces | 185 | 40 |
| fbPostTos | 4206 | 792 |
| fbUsers | 425 | 128 |
| fbUserAlbums | 14 | 8 |
| fbUserFriends | 11150 | 1808 |
| fbUserHometowns | 248 | 56 |
| fbUserLikes | 66477 | 13592 |
| fbUserLocations | 238 | 32 |
| fbUserPosts | 3145 | 504 |
| fbUserSignificantOthers | 69 | 8 |
| fbUserTaggedphotos | 109 | 8 |

Tabela 4.2: Količina prenesenih podatkov.

| oznaka meritve | opis |
|---|---|
| GET | čas med poslano zahtevo GET in prejetim odzivom Graph API-ja |
| CCH1 | čas, potreben za pridobitev ustreznega podatka iz Preslikovalne tabele oz. predpomnilnika |
| CCH2 | čas, potreben za pridobitev podatka o referenciranem objektu ustreznega polja `id` iz `DataContext`-a oz. predpomnilnika |
| CCH3 | čas, potreben za pridobitev seznama polj ustreznega objekta iz `DataContext`-a oz. predpomnilnika |
| PAG | čas procesiranja posameznega seznama, ki vsebuje več strani podatkov (ostranjevanje) |
| REF | čas procesiranja posameznega referenčnega objekta |
| TPG | čas izvajanja Niti za obdelavo strani |
| TPGW | čas čakanja pred kreacijo Niti za obdelavo strani v primeru prevelikega števila izvajajočih podrejenih niti |
| TRF | čas izvajanja Niti za obdelavo referenčnega objekta |
| TRFW | čas čakanja pred kreacijo Niti za obdelavo referenčnega objekta v primeru prevelikega števila izvajajočih podrejenih niti |
| PAGW | čas čakanja na obdelavo polja s seznamom objektov |
| SUM | celoten čas izvajanja algoritma |

Tabela 4.3: Izvedene meritve.

| nit | število |
|---|---|
| glavna nit algoritma | 1 |
| Nit za obdelavo strani | 713 |
| Nit za obdelavo referenčnega objekta | 5732 |
| skupaj | 6446 |

Tabela 4.4: Število kreiranih niti.

| izmerjeni čas (meritev SUM) | izračunani efektivni čas | predpomnenje | večnitenje |
|---|---|---|---|
| 1147,66 | 1784,16 | ✓ | ✓ |
| 1187,72 | 1187,72 | ✓ | × |
| 1678,56 | 5672,86 | × | ✓ |
| 1837,66 | 1837,66 | × | × |

Tabela 4.5: Izmerjeni celotni časi izvajanja algoritma (meritev SUM) in izračunani efektivni časi izvajanja niti algoritma glede na izbrano konfiguracijo.

| meritev | $N$ | $\sum t$ | $\bar{t} \pm \sigma$ | $min$ | $\tilde{t}$ | $max$ | pp. | vn. |
|---|---|---|---|---|---|---|---|---|
| GET | 1905 | 746,55 | $0,39 \pm 0,51$ | 0,09 | 0,16 | 4,56 | ✓ | ✓ |
| | | 755,34 | $0,40 \pm 0,51$ | 0,09 | 0,17 | 3,95 | ✓ | × |
| | | 685,14 | $0,36 \pm 0,47$ | 0,09 | 0,14 | 4,11 | × | ✓ |
| | | 744,00 | $0,39 \pm 0,52$ | 0,09 | 0,16 | 4,02 | × | × |
| CCH1 | 415252 | 5,47 | $0,0000 \pm 0,0005$ | 0,00 | 0,00 | 0,141 | ✓ | ✓ |
| | | 5,84 | $0,0000 \pm 0,0005$ | 0,00 | 0,00 | 0,141 | ✓ | × |
| | | 2740,91 | $0,0066 \pm 0,0106$ | 0,00 | 0,00 | 0,313 | × | ✓ |
| | | 691,66 | $0,0016 \pm 0,0050$ | 0,00 | 0,00 | 0,156 | × | × |
| CCH2 | 110161 | 0,70 | $0,0000 \pm 0,0003$ | 0,00 | 0,00 | 0,031 | ✓ | ✓ |
| | | 0,56 | $0,0000 \pm 0,0003$ | 0,00 | 0,00 | 0,016 | ✓ | × |
| | | 18,03 | $0,0002 \pm 0,0016$ | 0,00 | 0,00 | 0,063 | × | ✓ |
| | | 17,69 | $0,0002 \pm 0,0017$ | 0,00 | 0,00 | 0,094 | × | × |
| CCH3 | 104928 | 1,05 | $0,0000 \pm 0,0004$ | 0,00 | 0,00 | 0,016 | ✓ | ✓ |
| | | 1,00 | $0,0000 \pm 0,0004$ | 0,00 | 0,00 | 0,016 | ✓ | × |
| | | 70,30 | $0,0007 \pm 0,0034$ | 0,00 | 0,00 | 0,156 | × | ✓ |
| | | 67,97 | $0,0007 \pm 0,0032$ | 0,00 | 0,00 | 0,188 | × | × |
| PAG | 221 | 1781,55 | $8,06 \pm 50,81$ | 0,31 | 1,77 | 628,89 | ✓ | ✓ |
| | | 1555,50 | $7,04 \pm 55,40$ | 0,11 | 0,80 | 756,95 | ✓ | × |
| | | 3278,66 | $14,84 \pm 70,93$ | 0,48 | 5,14 | 748,19 | × | ✓ |
| | | 2611,09 | $11,81 \pm 91,37$ | 0,25 | 2,22 | 1308,73 | × | × |
| REF | 5732 | 1888,91 | $0,33 \pm 1,29$ | 0,02 | 0,14 | 50,23 | ✓ | ✓ |
| | | 866,28 | $0,15 \pm 0,70$ | 0,00 | 0,05 | 23,83 | ✓ | × |
| | | 3946,83 | $0,69 \pm 2,41$ | 0,05 | 0,20 | 74,94 | × | ✓ |
| | | 1544,70 | $0,27 \pm 1,24$ | 0,02 | 0,05 | 34,48 | × | × |
| TPG | 713 | 2633,81 | $3,69 \pm 5,56$ | 0,00 | 0,39 | 37,64 | ✓ | ✓ |
| | | 4920,05 | $6,90 \pm 6,98$ | 0,02 | 5,23 | 56,69 | × | ✓ |
| TPGW | 177 | 665,34 | $3,76 \pm 2,86$ | 1,25 | 2,50 | 12,52 | ✓ | ✓ |
| | 278 | 1385,08 | $4,98 \pm 3,95$ | 1,25 | 3,75 | 30,06 | × | ✓ |
| TRF | 5732 | 1891,39 | $0,33 \pm 1,29$ | 0,02 | 0,14 | 50,23 | ✓ | ✓ |
| | | 3948,56 | $0,69 \pm 2,41$ | 0,05 | 0,20 | 74,94 | × | ✓ |
| TRFW | 1853 | 2855,27 | $1,54 \pm 1,19$ | 1,25 | 1,25 | 17,59 | ✓ | ✓ |
| | 2091 | 4006,52 | $1,92 \pm 2,30$ | 1,25 | 1,25 | 41,38 | × | ✓ |
| PAGW | 181 | 368,06 | $2,03 \pm 6,69$ | 1,00 | 1,08 | 87,17 | ✓ | ✓ |
| | 197 | 774,98 | $3,93 \pm 8,57$ | 1,02 | 2,53 | 113,77 | × | ✓ |

Tabela 4.6: Rezultati meritev časovne zahtevnosti Algoritma za prenos podatkov. Oznaka *pp.* predstavlja predpomnenje, oznaka *vn.* pa večnitenje.

Algoritem je pri linernem izvajanju brez predpomnenja približno 40% celotnega izvajalnega časa stal in čakal na odzive Graph API-ja (meritev GET), z uporabo predpomnenja pa se je ta delež – zaradi bolj učinkovitega delovanja algoritma, medtem ko le-ta ne stoji – povečal še za 23 odstotnih točk. Omenjen čas čakanja nominalno sicer ni odvisen od parametrov, ki smo jih spreminjali v konfiguracijah algoritma, smo pa z uporabo večnitenja čakanje obšli, saj se v tem primeru algoritem efektivno izvaja, če vsaj ena nit ne čaka. Trdimo lahko, da je vsaka nit v povprečju med izvajanjem čakala na odzive približno 16% (brez predpomnenja) oz. 41% (s predpomnenjem) svojega efektivnega izvajalnega časa. Predpostavljamo, da je nominalni čas čakanja na odzive Graph API-ja odvisen od števila polj s seznami podatkov v odzivih ter stanja omrežne povezave in zasedenosti Facebookovih strežnikov.

Obhajanje omenjenega čakanja na odzive in vzporedno procesiranje referenčnih objektov prideta do izraza pri porabljenem času za obdelavo polj s seznami, ki uporabljajo ostranjevanje podatkov (meritev PAG). Iz opravljenih meritev zaradi možnosti podrejenih seznamov v objektih seznamov sicer ne moremo izračunati kolikšen delež celotnega izvajalnega časa se porabi za njihovo procesiranje, lahko pa trdimo, da se je izmerjen čas procesiranja seznamov relativno na efektivni izvajalni čas algoritma z vključitvijo večnitenja zmanjšal za 47% (brez predpomnenja) oz. 23% (s predpomnenjem). Ta sprememba nam pove, da algoritem sezname pričakovano procesira bolj učinkovito z uporabo večnitenja kot brez.

Algoritem je za procesiranje referenčnih objektov (meritev REF) pri linearnem izvajanju porabil približno 84% (brez predpomnenja) oz. 73% (s predpomnenjem) celotnega izvajalnega časa (meritev vključuje tudi čakanje na odzive Graph API-ja in procesiranje seznamov pri posameznem referenčnem objektu). Predpostavljamo, da je torej ostalih 16% oz. 27% porabil večinoma za obdelavo glavnega objekta `user`. Sklepamo, da je do različnih deležev med izvajanjem s predpomnenjem in brez njega prišlo zaradi večjega števila iskanja predpomnjenih podatkov med obdelavo referenčnih objektov kot med obdelavo glavnega objekta `user`. Glede na opravljene meritve težko rečemo koliko časa je algoritem za procesiranje referenčnih objektov porabil pri večnitnem delovanju.

Uporaba predpomnenja je algoritem občutno pospešila, in sicer za 32% oz. 35% – pri linearnem izvajanju nekoliko več kot pri večnitnem. Do pohitritve je prišlo zaradi hitrejšega iskanja podatkov o Preslikovalni tabeli in strukturi podatkovne baze (meritve CCH1, CCH2 in CCH3). Časovna razlika pri posameznem iskanju podatkov je bila sicer nominalno gledano zelo majhna (v tisočinkah sekunde), vendar je zaradi velikega števila

ponovitev (skupno 630341) v algoritmu prišla močno do izraza. S predpomnenjem smo pri linearnem izvajanju delež celotnega izvajalnega časa algoritma, porabljen za iskanje teh podatkov, zmanjšali z 42% na zanemarljivih 0,6%. Posamezna nit je pri nelinearnem izvajanju algoritma porabila za iskanje podatkov v primeru konfiguracije brez predpomnenja 65%, sicer pa zanemarljivih 0,4% svojega efektivnega izvajalnega časa. Pri večnitnem izvajanju brez predpomnenja so niti algoritma efektivno porabile za iskanje občutno več časa kot pri linearnem izvajanju, do česar je prišlo zaradi zaklepanja virov. Slednje omejuje iskanje podatkov po podatkovni bazi oz. objektu `DataContext` naenkrat le eni niti. Algoritem zaradi tega sicer nikoli ne stoji, saj se vedno izvaja vsaj ena nit, pride pa glede na vse niti skupaj do efektivno daljših čakanj.

Uporaba večnitenja je algoritem pohitrila za 3% (s predpomnenjem) oz. 9% (brez predpomnenja). Do pohitritve je prišlo zaradi že omenjenega obhajanja čakanja na odzive Graph API-ja in vzporednega procesiranja strani in referenčnih objektov oz. izvajanja algoritma na več – v našem primeru dveh – procesnih jedrih v primerjavi z enim. Opazili smo, da je bila pri večnitnem izvajanju algoritma mediana časov čakanja niti pred kreacijo podrejenih Niti za obdelavo referenčnega objekta (meritev TRFW) enaka izbranemu minimumu za to čakanje (Tabela 4.1). Iz tega lahko sklepamo, da so niti pred kreacijo podrejenih niti v večini primerov čakale dlje, kot je bilo potrebno, kar pomeni, da bi s skrajšanjem minimuma za čakanje lahko algoritem še dodatno pohitrili.

Uporaba predpomnenja in večnitenja skupaj je aplikacijo pohitrila za več kot tretjino (38%), iz česar sledi, da je bila implementacija obeh optimizacijskih tehnik dobra odločitev.

# Poglavje 5

# Zaključek

V diplomski nalogi smo z izdelavo aplikacije *ObrazoKnjigoGled* pokazali način izdelave preprostega uporabniškega vmesnika z uporabo osnovnih spletnih tehnologij HTML, CSS in JS, ki temelji na eni sami spletni strani in s pomočjo tehnik AJAX podaja tekočo uporabniško izkušnjo brez večkratnega nalaganja vsebine. Predstavili smo objekte, definirane v programskem jeziku C#, z uporabo katerih lahko na preprost način generiramo dinamično vsebino med izvajanjem aplikacije.

Opisali smo strukturo Facebookovega socialnega grafa, implementacijo avtentikacije uporabnikov preko Facebooka in način programskega dostopa do podatkov na Facebooku. Podrobno smo obravnavali razvit algoritem, namenjen prenosu podatkov z uporabniškega profila na Facebooku v lokalno podatkovno bazo, pri katerem smo uporabili nekatere zahtevnejše tehnike programiranja, kot so predpomnenje, večnitenje in dinamično programiranje z uporabo generičnih metod, refleksije in dinamičnih objektov. Izvedli in predstavili smo rezultate meritev izvajalnega časa omenjenega algoritma v odvisnosti od uporabe predpomnenja in večnitenja ter zapisali naše ugotovitve o njegovem delovanju. Pri raziskavi Facebookovega spletnega vmesnika za programiranje smo opazili površnosti v dokumentaciji in nekonsistentnost v odzivih.

Prikazali smo nekatere možnosti analize uporabniških podatkov s Facebooka in načine, kako uporabniku predstaviti nove informacije z uporabo dveh načinov izrisa vizualizacij, in sicer z vmesnikom Google Charts in knjižnico Data-Driven Documents. Opisali smo metode za izračune povezane s časom in lokacijo točk na Zemlji in obratno geokodiranje koordinat z uporabo spletnega vmesnika in lokalne implementacije.

Z uporabo večslojne aplikacijske arhitekture, razdelitvijo projektnih knjižnic v funkcionalne sklope in upoštevanjem principa DRY smo izdelali obvladljivo in neponavljajočo

kodo z možnostjo ponovne uporabe pri nadaljnjem delu in razvoju novih aplikacij.

Čeprav je razvita spletna aplikacija namenjena uporabi več uporabnikom hkrati, nam je zaradi časovne omejitve diplomskega dela ni uspelo postaviti iz razvojnega okolja v produkcijsko in jo opazovati med izvajanjem v večuporabniškem okolju.

## 5.1 Nadaljnje delo

Poleg uporabe v realnem večuporabniškem okolju predstavljena spletna aplikacija omogoča še veliko možnosti v smeri nadaljnjega razvoja in izboljšav.

Knjižnico za registracijo in avtentikacijo uporabnikov (Poglavje 3.4) bi lahko nadgradili z uporabo piškotkov (angl. *cookies*). S tem se uporabnikom ne bi bilo potrebno avtenticirati preko Facebooka ob vsakem ponovnem zagonu spletnega brskalnika. Implementirali bi lahko tudi uporabniške skupine, npr. administratorje, uporabnike in goste ter v zaledni logiki generirali specifično vsebino glede na skupino, v katero uporabnik spada. Podprli bi lahko tudi možnost preklica določenih dovoljenj aplikacije za dostop do podatkov.

Z uporabo naprednejših algoritmov za analizo podatkov bi lahko odkrili in vključili v Poročilo (Poglavje 3.6.3) še marsikaj. Sekcijo o mreži prijateljev bi lahko npr. nadgradili z implementacijo algoritma za razvrščanje v skupine (angl. *clustering algorithm*). Z analizo pridobljenih skupin bi lahko nato uporabniku prikazali, kaj imajo njegovi prijatelji v posamezni skupini skupnega. Pri generaciji Poročila bi lahko uporabniku podali več kontrole nad prikazom želenih informacij. Primer takšne funkcionalnosti bi predstavljal spustni seznam na Sekciji o prijateljih, s katerim bi uporabnik izbral želeno skupino prijateljev, izdelano na Facebooku ali v aplikaciji *OKG*, katerih podatki bi se upoštevali pri analizi podatkov. Uporabnikom bi lahko omogočili tudi možnost izvoza svojih podatkov in/ali Poročila, v primeru večuporabniškega delovanja pa bi lahko razvili še generacijo Poročila z analizo podatkov vseh uporabnikov v podatkovni bazi skupaj.

Izboljšali bi lahko uporabniško izkušnjo z oblikovanjem lepšega in bolj intuitivnega dizajna spletne strani aplikacije ter vključitvijo namigov o uporabi aplikacije na uporabniškem vmesniku. Ker smo aplikacijo testirali le z uporabo spletnega brskalnika Google Chrome, bi v poštev prišla še optimizacija uporabniškega vmesnika za ostale brskalnike in mobilne naprave.

Algoritem za prenos podatkov smo implementirali z uporabo dinamičnega programiranja, kar pomeni, da bi ga lahko z minimalnimi spremembami uporabili tudi za prenos podatkov z drugih spletnih API-jev, ki vračajo podatke v obliki JSON. Algoritem vse

pomembne parametre za izvajanje sproti bere s podatkovne baze, torej bi morali v tem primeru izdelati novo podatkovno bazo, ki bi posnemala odzivne podatke želenega API-ja.

Trenutna implementacija podatkovnih komponent aplikacije zaradi uporabe LINQ-to-SQL-a zahteva pri vsaki spremembi strukture podatkovne baze posodobitev objektov `DataContext`. Posledično je za pravilno delovanje algoritmov nato potrebna ponovna graditev (angl. *build*) aplikacije. Ponovnim graditvam ob takšnih spremembah bi se lahko izognili s pametno uporabo ADO.NET-a namesto LINQ-to-SQL-a, pri čemer bi dinamično generirali poizvedbe SQL med izvajanjem aplikacije, predpomnjene informacije o strukturi podatkovne baze pa bi intervalno osveževali.

# Literatura


[1] N. Carlson. "At Last – The Full Story Of How Facebook Was Founded", *Business Insider* (online). 5. marec 2010 (citirano 8. septembra 2013). Dostopno na naslovu: http://www.businessinsider.com/how-facebook-was-founded-2010-3

[2] Facebook, Inc. "Facebook Reports First Quarter 2013 Results", *Facebook Investor Relations* (online). Menlo Park, CA, ZDA, *PR Newswire*, 1. maj 2013 (citirano 8. septembra 2013). Dostopno na naslovu: http://investor.fb.com/releasedetail.cfm?ReleaseID=761090

[3] C. Abram, L. Pearlman, "Facebook for Dummies", *For Dummies (4th ed.). John Wiley and Sons*, 2008.

[4] S. Wolfram. "Wolfram Alpha Personal Analytics for Facebook", *Wolfram Alpha Blog* (online). 30. avgust 2012 (citirano 8. septembra 2013). Dostopno na naslovu: http://blog.wolframalpha.com/2012/08/30/↩
wolframalpha-personal-analytics-for-facebook

[5] A. Troelsen. "Dynamic Types and the Dynamic Language Runtime", *Pro C# 2010 and the .NET 4 Platform. Apress*, 2010, s. 701-724.

[6] A. Rusina. "Understanding the Dynamic Keyword in C# 4" (online), *MSDN Magazine (February 2011 Issue). Microsoft*, februar 2011. Dostopno na naslovu: http://msdn.microsoft.com/en-us/magazine/gg598922.aspx

[7] A. Freeman, J. C. Rattz Jr. "LINQ to SQL Introduction", *Pro LINQ. Apress*, 2010, s. 437-447.

[8] D. Kulkarni, L. Bolognese, M. Warren, A. Hejlsberg, K. George. "LINQ to SQL: .NET Language-Integrated Query for Relational Data", *Microsoft Developer Network*

(online). *Microsoft*, marec 2007 (citirano 8. septembra 2013). Dostopno na naslovu: http://msdn.microsoft.com/en-us/library/bb425822.aspx

[9] A. Troelsen. "ADO.NET Part II: The Disconnected Layer", *Pro C# 2010 and the .NET 4 Platform. Apress*, 2010, s. 885-950.

[10] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures", *doktorska disertacija*. Irvine, CA, ZDA, *University of California*, 2000.

[11] A. Freeman. "Working with Ajax", *Applied ASP .NET 4 in Context. Apress*, 2011, s. 241-275.

[12] V. Gopalakrishnan. "IIS Express Overview", *ISS.NET* (online). *Microsoft*, julij 2010 (citirano 8. septembra 2013). Dostopno na naslovu: http://www.iis.net/learn/extensions/introduction-to-iis-express/iis-express-overview

[13] Moo Nam Ko, G. P. Cheek, M. Shehab, R. Sandhu. "Social-Networks Connect Services", *IEEE Computer (Vol. 43)*. 2010, 8:37-43.

[14] Facebook, Inc. "Facebook Platform Launches", *Facebook Developers Blog* (online). 27. maj 2007 (arhivirano 27. maja 2007, citirano 8. septembra 2013). Dostopno na naslovu: http://web.archive.org/web/20110522075406/↩ http://developers.facebook.com/blog/post/21

[15] Facebook, Inc. "Documentation", *Facebook Developers* (online). (Poso- dobljeno 9. septembra 2013, citirano 14. septembra 2013). Dostopno na naslovu: https://developers.facebook.com/docs

[16] Na Wang, Xu Heng, J. Grossklags. "Third-party apps on Facebook: privacy and the illusion of control", *Proceedings of the 5th ACM Symposium on Computer Human Interaction for Management of Information Technology (CHIMIT '11)*. New York, NY, ZDA, *ACM*, 2011, 4:1-10.

[17] J. Ugander, B. Karrer, L. Backstrom, C. Marlow. "The Anatomy of the Facebook Social Graph", *CoRR (abs/1111.4503)*. 2011.

[18] M. Wolf, C. Wicksteed. "Date and time formats", *W3C NOTE NOTE-datetime-19980827* (online). 1999 (citirano 20. septembra 2013). Dostopno na naslovu: http://www.w3.org/TR/NOTE-datetime

[19] A. Hunt, D. Thomas. "The pragmatic programmer: from journeyman to master". *Addison-Wesley*, 2000.

[20] M. Fowler. "Patterns of Enterprise Application Architecture". Boston, MA, *Addison-Wesley*, 2002.

[21] W. W. Eckerson. "Three Tier Client/Server Architectures: Achieving Scalability, Performance, and Efficiency in Client/Server Applications", *Open Information Systems (Vol. 3)*. Januar 1995, 20:46-50.

[22] A. Mesbah. "Analysis and Testing of Ajax-based Single-page Web Applications", *doktorska disertacija*. 2009.

[23] A. Mesbah, A. van Deursen. "Migrating Multi-page Web Applications to Single-page AJAX Interfaces", *11th European Conference on Software Maintenance and Reengineering CSMR*. 2007, s. 181-190.

[24] J. Albahari. "Threading in C#" (online). *Joseph Albahari & O'Reilly Media, Inc.*, 2006 (posodobljeno 27. aprila 2011, citirano 20. septembra 2013). Dostopno na naslovu:
http://hfs1.duytan.edu.vn/upload/ebooks/3758.pdf

[25] J. Albahari, B. Albahari. "Threading", *C# 4.0 in a Nutshell*. 3. februar 2010, s. 831-918.

[26] J. Albahari, B. Albahari. "try Statements and Exceptions", *C# 4.0 in a Nutshell*. 3. februar 2010, s. 141-150.

[27] Google, Inc. "The Google Geocoding API", *Google Developers* (online). (Posodobljeno 10. septembra 2013) (citirano 22. septembra 2013). Dostopno na naslovu:
https://developers.google.com/maps/documentation/geocoding

[28] W. C. Forsythe, E. J. Rykiel Jr., R. S. Stahl, H. Wu, R. M. Schoolfield. "A model comparison for daylength as a function of latitude and day of year", *Ecological Modelling (Vol. 80). Elsevier*, 1995, 1:87-95.

[29] R. W. Sinnott. "Virtues of the Haversine", *Sky and Telescope (Vol. 68)*. December, 1984, 2:158.

[30] J. Josekutty, G. Elert. "Speed of a Commercial Jet Airplane", *The Physics Factbook*$^{TM}$ (online). 2002 (citirano 22. septembra 2013). Dostopno na naslovu: http://hypertextbook.com/facts/2002/JobyJosekutty.shtml

# Dodatek

## Objekti Facebookovega socialnega grafa

Sledijo tabele z opisi polj in povezav objektov Facebookovega socialnega grafa. Navedene so le informacije, ki smo jih potrebovali pri razvoju aplikacije *ObrazoKnjigoGled*.

| polje | tip polja | opis |
|---|---|---|
| `id` | preprosto | identifikacijska številka |
| `name` | preprosto | ime |
| `category` | preprosto | kategorija |
| `location` | kompleksno | lokacija s preprostima poljema `latitude` in `longitude` |

Tabela 1: Polja in povezave objekta `page`, ki predstavlja javni profil.

| polje | tip polja | opis |
|---|---|---|
| `id` | preprosto | identifikacijska številka |
| `name` | preprosto | ime |
| `picture` | preprosto | javni URL slike |
| `height` | preprosto | višina v pikslih |
| `width` | preprosto | dolžina v pikslih |
| `created_time` | preprosto | datum kreacije |
| `place` | kompleksno | lokacija v obliki objekta `page` s preprostima poljema `id` in `name` ter kompleksnim poljem `location` s poljema `latitude` in `longitude` |
| `from` | kompleksno | objekt, ki je objavil sliko s poljema `id` in `name` |
| `tags` | povezava | objekti, ki so označeni na sliki, s poljem `created_time` |
| `likes` | povezava | objekti, ki so všečkali sliko, s poljem `created_time` |
| `comments` | povezava | komentarji v obliki objektov `comment` |

Tabela 2: Polja in povezave objekta `photo`, ki predstavlja sliko.

| polje | tip polja | opis |
|---|---|---|
| `id` | preprosto | identifikacijska številka |
| `name` | preprosto | ime |
| `message` | preprosto | sporočilo |
| `created_time` | preprosto | datum kreacije |
| `from` | kompleksno | objekt, ki je objavil sliko, s poljema `id` in `name` |

Tabela 3: Polja in povezave objekta `comment`, ki predstavlja komentar.

| polje | tip polja | opis |
|---|---|---|
| `id` | preprosto | identifikacijska številka |
| `name` | preprosto | ime |
| `type` | preprosto | tip |
| `created_time` | preprosto | datum kreacije |
| `photos` | povezava | slike v obliki objektov `photo` |
| `likes` | povezava | objekti, ki so všečkali album, s poljem `created_time` – časom všečkanja |
| `comments` | povezava | komentarji v obliki objektov `comment` |

Tabela 4: Polja in povezave objekta `album`, ki predstavlja album s slikami.

| polje | tip polja | opis |
|---|---|---|
| `id` | preprosto | identifikacijska številka |
| `created_time` | preprosto | datum kreacije |
| `status_type` | preprosto | tip objave |
| `message` | preprosto | sporočilo |
| `story` | preprosto | tekst objave, ki ga ni namerno generiral uporabnik |
| `story_type` | preprosto | tip parametra `story` |
| `place` | kompleksno | lokacija v obliki objekta `page` s preprostima poljema `id` in `name` ter kompleksnim poljem `location` s poljema `latitude` in `longitude` |
| `from` | kompleksno | objekt, ki je objavil sliko, s poljema `id` in `name` |
| `to` | kompleksno | objekt, kateremu je bila objava namenjena, s poljema `id` in `name` |
| `likes` | povezava | objekti, ki so všečkali objavo, s poljem `created_time` – časom všečkanja |
| `comments` | povezava | komentarji v obliki objektov `comment` |

Tabela 5: Polja in povezave objekta `post`, ki predstavlja objavo.

| | | | |
|---|---|---|---|
| `id` | preprosto | identifikacijska številka | |
| `name` | preprosto | polno ime | |
| `birthday` | preprosto | datum rojstva | `user_birthday`, `friends_birthday` |
| `first_name` | preprosto | ime | |
| `gender` | preprosto | spol | |
| `last_name` | preprosto | priimek | |
| `middle_name` | preprosto | drugo ime | |
| `relationship_status` | preprosto | zakonski stan | `user_relationships`, `friends_relationships` |
| `timezone` | preprosto | časovni pas | `access_token` |
| `hometown` | kompleksno | rojstni kraj v obliki objekta `page` s poljema `id` in `name` | `user_hometown`, `friends_hometown` |
| `location` | kompleksno | lokacija v obliki objekta `page` s poljema `id` in `name` | `user_location`, `friends_location` |
| `significant_other` | kompleksno | boljša polovica v obliki objekta `user` s poljema `id` in `name` | `user_relationships`, `friends_relationships` |
| `friends` | povezava | prijatelji v obliki objektov `user` | `access_token` |
| `likes` | povezava | všečki v obliki objektov `page` | `user_likes`, `friends_likes` |
| `albums` | povezava | albumi slik v obliki objektov `album` | `user_photos`, `friends_photos` |
| `photos` | povezava | objavljene slike v obliki objektov `photo`; vključuje tudi slike drugih uporabnikov, v katerih je uporabnik označen | `user_photos`, `friends_photos` |
| `feed` | povezava | vse objave na profilu v obliki objektov `post` | `read_stream` |

Tabela 6: Polja in povezave objekta `user`, ki predstavlja uporabnika oz. njegov profil. Tabela vključuje dovoljenja, potrebna za dostop do polj in povezav preko Graph APIja.

# Vzorci kode

Sledi nekaj vzorcev kode, vključno z izvorno kodo spletne aplikacije *ObrazoKnjigoGled* s pripadajočimi knjižnicami. Zaradi preglednosti in vsebine diplomske naloge smo določene elemente originalne aplikacijske izvorne kode tu poenostavili.

```
function GetContentFromServer() {
    $.ajax({
        type: "POST",
        url: ajaxServiceUrl + '/GetContent',
        contentType: "application/json; charset=utf-8",
        dataType: "json",
        success: function (msg) {
            if (msg == null || msg.d == null) { return; }
            ProcessJson(msg.d);
        }
    });
}

function ProcessJson(jsonString) {
    var jsonData = JSON.parse(jsonString);
    // Process Html.
    var html = jsonData['Html'];
    for (var i = 0; i < html.length; i++) {
        var container = $(html[i].Selector);
        container.html(html[i].Code);
    }
    // Process JavaScript.
    var javascript = jsonData['JavaScript'];
    for (var j = 0; j < javascript.length; j++) {
        var newScriptTag = document.createElement('script');
        newScriptTag.appendChild( document.createTextNode(javascript[j].Code));
        document.body.appendChild(newScriptTag);
    }
}
```

Vzorec kode 1: JS – primera metod, izmed katerih prva pokliče metodo za generacijo dinamične vsebine na strežniku (Vzorec kode 2) in prejeto vsebino pošlje drugi, ki jo vstavi v spletno stran.

```
[OperationContract]
public static string GetContent()
{
    var content = new JsonContent();
    content.CreateHtmlBlock("#someContainerId",
        "<p>Hello world!</p>" +
        "<p><a class='clickable'>Click here.</a></p>");
    content.CreateJavascriptBlock(@"
        $('#someContainerId a.clickable').click(function () {
            ⋮
        });
    ");
    return content.ToString();
}
```

Vzorec kode 2: C# – primer metode, ki ob zahtevi GET generira in vrne dinamično vsebino. Objekt `JsonContent` je definiran v Vzorcu kode 3.

```
public class JsonContent : JsonSerializable
{
    private List<JsonHtmlBlock> _html = new List<JsonHtmlBlock>();
    private List<JsonJavascriptBlock> _javaScript = new ↩
        List<JsonJavascriptBlock>();

    public JsonHtmlBlock[] Html { get { return _html.ToArray(); } }
    public JsonJavascriptBlock[] JavaScript
    {
        get { return _javaScript.ToArray(); }
    }

    public JsonHtmlBlock CreateHtmlBlock(string selector, string code)
    {
        var item = new JsonHtmlBlock(selector, code);
        _html.Add(item);
        return item;
    }

    public JsonJavascriptBlock CreateJavascriptBlock(string code)
    {
        var item = new JsonJavascriptBlock(code);
        _javaScript.Add(item);
        return item;
    }
}
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
public class JsonHtmlBlock : JsonSerializable
{
    public string Code { get; private set; }
    public string Selector { get; private set; }

    public JsonHtmlBlock(string selector, string code)
    {
        Selector = selector;
        Code = code;
    }
}

public class JsonJavascriptBlock : JsonSerializable
{
    public string Code { get; private set; }

    public JsonJavascriptBlock(string code)
    {
        Code = code;
    }
}

public abstract class JsonSerializable
{
    public override string ToString()
    {
        var jss = new JavaScriptSerializer();
        var result = jss.Serialize(this);
        return result;
    }
}
```

Vzorec kode 3: C# – primeri objektov, ki se ob klicu metode `ToString` serializirajo v niz znakov JSON.

```
public class PieChart : GoogleChart
{
    protected override string JavaScriptFilePath
    {
        get { return ... + "PieChart.js"); }
    }
    protected override GoogleChartOptions DefaultOptions
    {
        get
        {
            return new GoogleChartOptions
            {
                ChartArea = new GChartArea(0, 10, 100.0, 100.0),
                Colors = new GColors("#3366cc", "#dc3912", ...) //Default ↩
                    Google colors.
            };
        }
    }

    public PieChart(string id, string[] columns) : base(id, columns)
    {
        if (columns.Length != 2) throw new ↩
            ArgumentOutOfRangeException("Invalid column number.");
    }

    private void SetData(Dictionary<string, int> data)
    {
        StringBuilder sb = new StringBuilder();
        var anchor = "['{0}', {1}],\n\r";
        foreach (var pair in data)
            sb.AppendFormat(anchor, pair.Key.Replace("'", "\\'"), pair.Value);
        var dataStr = sb.ToString().TrimEnd().TrimEnd(',')

        JSBuilder.Replace("[DATA]", dataStr);
    }
}
```

Vzorec kode 4: C# – objekt `PieChart`, ki predstavlja krožni grafikon. Izvorna koda datoteke `PieChart.js` je navedena v Vzorcu kode 5, objekt `GoogleChartOptions` v Vzorcu kode 6, podedovan objekt `GoogleChart` pa v Vzorcu kode 8.

```
var [CHART] = { chart: null };

var [CHART]Options = {
    [COLORS],
    [CHARTAREA],
    fontSize: 13,
    fontName: 'Bookman Old Style',
    backgroundColor: '#303030',
    pieSliceText: 'percentage',
    pieSliceTextStyle: { color: '#eeeeee' },
    pieSliceBorderColor: '#303030',
    legend: { position: 'right', alignment: 'center', ... }
};

function Draw[CHART]() {
    var data = google.visualization.arrayToDataTable([
        [COLS],
        [DATA]
    ]);

    if (![CHART].chart) {
        [CHART].chart = new google.visualization ↩
            .PieChart(document.getElementById('[CHART]'));
        [CHART].chart.draw(data, [CHART]Options);
    }
}
```

Vzorec kode 5: JS – datoteka `PieChart.js`, ki vsebuje kodo za izris in interakcijsko funkcionalnost krožnega grafikona. Deli kode, zapisani v oglatih oklepajih, se zgenerirajo dinamično z uporabo objekta `PieChart` (Vzorec kode 4). Dokumentacija Googlovega krožnega grafikona, ki določa obliko te datoteke, je dostopna na naslovu: https://developers.google.com/chart/interactive/docs/↩ gallery/piechart

```
public class GoogleChartOptions : ChartOptions
{
    public GChartArea ChartArea;
    public GColors Colors;
}

public abstract class ChartOptions { }

public abstract class GoogleChartOption : ChartOption
{
    public abstract override string Key { get; }
    public abstract override string ToString();
}

// E.g. colors: ['#00ff00', '#ff0000']
public class GColors : GoogleChartOption
{
    public string[] Value { get; private set; }
    public override string Key { get { return "[COLORS]"; } }

    public GColors(params string[] value)
    {
        Value = value;
    }

    public override string ToString()
    {
        return string.Format("colors: [{0}]",
            string.Concat(
                Value.Select(x => string.Format("'{0}', ", x))
            ).TrimEnd().TrimEnd(','));
    }
}

// E.g. chartArea: {left: 89, top: 15, width: '90%', height: '80%'}
public class GChartArea : GoogleChartOption
{
    public string Top { get; private set; }
    public string Left { get; private set; }
    public string Width { get; private set; }
    public string Height { get; private set; }
    public override string Key { get { return "[CHARTAREA]"; } }
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
    public GChartArea(int t, int l, double w, double h)
    {
        Top = string.Format("{0}", t);
        Left = string.Format("{0}", l);
        Width = string.Format("'{0:0.000}%'", w);
        Height = string.Format("'{0:0.000}%'", h);
    }

    public override string ToString()
    {
        return string.Format("chartArea: {{left: {0}, top: {1}, width: {2}, ↩
            height: {3}}}",
            Left, Top, Width, Height);
    }
}

public abstract class ChartOption
{
    public abstract string Key { get; }
    public abstract override string ToString();
}
```

Vzorec kode 6: C# – objekti, ki predstavljajo nastavitvene elemente Googlovih vizualizacij, razvidne iz dokumentacije API-ja Google Charts.

```
public abstract class Chart<TOptions, TOption> : Chart
    where TOptions : ChartOptions, new()
    where TOption : ChartOption
{
    protected TOptions _Options;
    public TOptions Options
    {
        get
        {
            if (_Options == null)
                _Options = new TOptions();
            return _Options;
        }
    }
    protected abstract TOptions DefaultOptions { get; }
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
    public override string JavaScript
    {
        get
        {
            var result = new StringBuilder(JSBuilder.ToString());
            SetOptions(result);
            return result.ToString();
        }
    }

    public Chart(string id) : base(id) { }

    protected void SetOptions(StringBuilder sb)
    {
        var options = typeof(TOptions)
            .GetFields(BindingFlags.Public | BindingFlags.Instance);
        foreach (FieldInfo field in options)
        {
            var value = (field.GetValue(Options) ?? ↩
                field.GetValue(DefaultOptions)) as TOption;
            if (value != null)
                sb.Replace(value.Key, value.ToString());
        }
    }
}

public abstract class Chart
{
    private StringBuilder _javaSb;

    public string Id { get; protected set; }

    protected abstract string JavaScriptFilePath { get; }
    public abstract string JavaScript { get; }
    protected StringBuilder JSBuilder
    {
        get
        {
            if (_javaSb == null)
                _javaSb = new StringBuilder(
                    File.ReadAllText(JavaScriptFilePath));
            return _javaSb;
        }
    }
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
    public Chart(string id)
    {
        this.Id = id;
        JSBuilder.Replace("[CHART]", id);
    }
}
```

Vzorec kode 7: C# – abstraktni in abstraktno-generični objekt `Chart`, ki predstavljata osnovo vizualizacije. Objekta `ChartOptions` in `ChartOption` sta definirana v Vzorcu kode 6.

```
public abstract class GoogleChart
    : Chart<GoogleChartOptions, GoogleChartOption>
{
    protected abstract override string JavaScriptFilePath { get; }
    protected abstract override GoogleChartOptions DefaultOptions { get; }

    public GoogleChart(string id, string[] columns) : base(id)
    {
        SetCols(columns);
    }

    private void SetCols(string[] columns)
    {
        StringBuilder colsSb = new StringBuilder("[");

        for (int i = 0; i < columns.Length; i++)
            colsSb.AppendFormat("'{0}',", columns[i]);
        string colsStr = colsSb.ToString().TrimEnd(',') + "]";

        JavaScriptBuilder.Replace("[COLS]", colsStr);
    }
}
```

Vzorec kode 8: C# – abstraktni objekt `GoogleChart`, ki predstavlja temelj vizualizacije, ki za prikaz in interakcijsko funkcionalnost uporablja API Google Charts. Podedovan objekt `Chart` je definiran v Vzorcu kode 7.

```
public sealed class ChartBox<T> : ChartBox, IList<T>, ICollection<T>, ↩
    IEnumerable<T>, IEnumerable where T : Chart
{
    private List<T> _charts = new List<T>();

    #region IList, ICollection & IEnumerable Implementation
    // Code which handles getting, adding, and removing of Chart objects ↩
        from/to the _charts list.
    ⋮
    #endregion

    public override string ToJavaScript()
    {
        var sb = new StringBuilder();
        foreach (var chart in _charts)
        {
            sb.Append(chart.JavaScript);
            if (!Style.DrawOnUserAction)
                sb.Append(string.Format("Draw{0};", chart.Id));
        }
        return sb.ToString();
    }

    public override HtmlTag ToHtml()
    {
        // Create the root Html element for the ChartBox.
        var root = new HtmlTag("div", true, null);
        root.AddClass("chartbox");

        // Add custom CSS classes.
        foreach(var cssClass in Style.Classes)
            root.AddClass(cssClass);

        // Add the title.
        var title = root.AddChild(new HtmlTag("div", true, Style.Title));
        title.AddClass("title");

        // Set chart toggles.
        if (Style.Toggles != null && Style.Toggles.Length > 0)
        {
            var tag = new HtmlAnchor(Style.Toggles[0]);
            tag.AddClass("toggle");
            title.Value = string.Format("{0} ({1})", title.Value, tag);
        }
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
        // Set info text.
        if (Style.InfoText != null)
        {
            var tag = title.AddChild(new HtmlTag("a", true, "?"));
            tag.AddClass("infotext");
            tag.AddChild(new HtmlTag("div", true, Style.InfoText));
        }
        // Set extra text.
        if (Style.ExtraText != null)
        {
            var tag = title.AddChild(new HtmlTag("a", true, "*"));
            tag.AddClass("extratext");
            tag.AddChild(new HtmlTag("div", true, Style.ExtraText));
        }

        // Add a vizualize now button.
        if (Style.DrawOnUserAction) { ... }

        // Add chart roots (charts are generated by JavaScript).
        foreach (var chart in _charts)
        {
            var tag = root.AddChild(new HtmlTag("div", true, null));
            tag.AddClass("chart");
            tag.SetId(chart.Id);
        }

        return root;
    }
}

public abstract class ChartBox : IHtmlable
{
    public ChartBoxStyle Style;
    public abstract HtmlTag ToHtml();
    public abstract string ToJavaScript();
}
```

Vzorec kode 9: C# – generični objekt `ChartBox`, ki vsebuje vizualizacije v obliki objektov `Chart`, s pripadajočo abstraktno osnovo. Vmesnik `IHtmlable` in struktura `ChartBoxStyle` sta definirana v Vzorcu kode 10, objekt `HtmlTag` pa v Vzorcu kode 11.

```
public interface IHtmlable
{
    HtmlTag ToHtml();
}

public struct ChartBoxStyle
{
    public string Title;
    public string InfoTextFormat;
    public string ExtraText;
    public string[] Toggles;
    public string[] Classes;
    public bool DrawOnUserAction;
}
```

Vzorec kode 10: C# – Vmesnik `IHtmlable` in struktura `ChartBoxStyle`.

```
public class HtmlTag : XmlTag
{
    private string _id;
    private List<string> _classes = new List<string>();
    private Dictionary<string, string> _styles = new Dictionary<string, ↩
        string>();

    public string Id { get { return _id; } }
    public string[] Classes { get { return _classes.ToArray(); } }
    public KeyValuePair<string, string>[] Styles { get { return ↩
        _styles.ToArray(); } }

    public HtmlTag(string name, bool isExpandable, object value)
        : base(name, isExpandable, value) { }

    public void SetId(string value)
    {
        var idAttr = base.GetAttribute("id");
        if (idAttr == null)
            idAttr = new XmlAttribute("id", value);
        else
            idAttr.Value = value; ;
        base.SetAttribute(idAttr);
    }
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
public void RemoveId()
{
    base.RemoveAttribute("id");
}
public void AddClass(string item)
{
    if (_classes.Contains(item)) return;
    _classes.Add(item);
    RefreshClasses();
}

public bool RemoveClass(string item)
{
    if (_classes.Remove(item))
    {
        RefreshClasses();
        return true;
    }
    return false;
}
private void RefreshClasses()
{
    var attr = base.GetAttribute("class") ?? new XmlAttribute("class");
    attr.Value = string.Concat(
            _classes.Select(x => string.Format("{0} ", x))
        ).TrimEnd();
    base.SetAttribute(attr);
}

public void SetStyle(string key, object value)
{
    _styles[key] = value.ToString();
    RefreshStyles();
}
public bool RemoveStyle(string key)
{
    if (_styles.Remove(key))
    {
        RefreshStyles();
        return true;
    }
    return false;
}
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
    private void RefreshStyles()
    {
        var attr = base.GetAttribute("style")
            ?? new XmlAttribute("style");
        attr.Value = string.Concat(
                _styles.Select(x => string.Format("{0}: {1}; ", x.Key, x.Value))
            ).TrimEnd();
        base.SetAttribute(attr);
    }

    public override XmlAttribute GetAttribute(string name)
    {
        if (name == "id") throw new ArgumentException("Use the id-related ↩
            methods to handle the id.");
        if (name == "class") throw new ArgumentException("Use the ↩
            class-related methods to handle classes.");
        if (Name == "style") throw new ArgumentException("Use the ↩
            style-related methods to handle styles.");
        return base.GetAttribute(name);
    }
    public override void SetAttribute(XmlAttribute item)
    {
        if (item.Name == "id") throw new ArgumentException("Use the ↩
            id-related methods to handle the id.");
        if (item.Name == "class") throw new ArgumentException("Use the ↩
            class-related methods to handle classes.");
        if (item.Name == "style") throw new ArgumentException("Use the ↩
            style-related methods to handle styles.");
        base.SetAttribute(item);
    }
    public override bool RemoveAttribute(XmlAttribute item)
    {
        if (item.Name == "id") throw new ArgumentException("Use the ↩
            id-related methods to handle the id.");
        if (item.Name == "class") throw new ArgumentException("Use the ↩
            class-related methods to handle classes.");
        if (item.Name == "style") throw new ArgumentException("Use the ↩
            style-related methods to handle styles.");
        return base.RemoveAttribute(item);
    }
}
```

Vzorec kode 11: C# – objekt `HtmlTag`, ki predstavlja značko HTML in se ob klicu metode `ToString` vanjo serializira. Podedovan objekt `XmlTag` je definiran v Vzorcu kode 12, objekt `XmlAttribute` pa v Vzorcu kode 13.

```
public class XmlTag
{
    public string Name { get; protected set; }
    public object Value { get; set; }
    public bool IsExpandable { get; protected set; }

    private XmlTag _parent;
    public XmlTag Parent { get { return _parent; } }

    private List<XmlAttribute> _attributes = new List<XmlAttribute>();
    public XmlAttribute[] Attributes { get { return _attributes.ToArray(); } }

    private List<XmlTag> _children = new List<XmlTag>();
    public XmlTag[] Children { get { return _children.ToArray(); } }

    public XmlTag(string name, bool isExpandable, object value)
    {
        Name = name.Trim();
        IsExpandable = isExpandable;
        if (value is XmlTag)
            AddChild(value as XmlTag);
        else
            Value = value;
    }

    public virtual XmlAttribute GetAttribute(string name)
    {
        return _attributes.Where(x => x.Name == name).SingleOrDefault();
    }
    public void SetAttribute(string name, object value)
    {
        SetAttribute(new XmlAttribute(name, value));
    }
    public virtual void SetAttribute(XmlAttribute item)
    {
        for (int i = 0; i < _attributes.Count; i++)
            if (_attributes[i].Name == item.Name)
            {
                _attributes[i] = item;
                return;
            }
        _attributes.Add(item);
    }
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
public bool RemoveAttribute(string name)
{
    return RemoveAttribute(new XmlAttribute(name));
}
public virtual bool RemoveAttribute(XmlAttribute item)
{
    for (int i = 0; i < _attributes.Count; i++)
        if (_attributes[i].Name == item.Name)
        {
            _attributes.RemoveAt(i);
            return true;
        }
    return false;
}

public virtual XmlTag GetFirstChild()
{
    return _children.FirstOrDefault();
}
public virtual XmlTag GetFirstChild(string name)
{
    return _children.FirstOrDefault(x => x.Name == name);
}
public virtual XmlTag[] GetChildren(string name)
{
    return _children.Where(x => x.Name == name).ToArray();
}
public virtual T AddChild<T>(T item) where T : XmlTag
{
    _children.Add(item);
    item._parent = this;
    return item;
}

public virtual bool RemoveChild(XmlTag item)
{
    for (int i = 0; i < _attributes.Count; i++)
        if (_children[i] == item)
        {
            _children.RemoveAt(i);
            return true;
        }
    return false;
}
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
    public override string ToString()
    {
        var sb = new StringBuilder();
        sb.AppendFormat("<{0}", Name);
        _attributes.ForEach(x => sb.Append(' ' + x.ToString()));

        sb.Append(IsExpandable ? ">" : " />");
        if (IsExpandable)
        {
            if (Value != null)
                sb.Append(Value.ToString());

            if (_children.Count > 0)
            {
                sb.Append("\n");
                _children.ForEach(x => sb.AppendFormat("\t{0}", x.ToString()));
            }
            sb.AppendFormat("</{0}>", Name);
        }

        sb.Append("\n");
        return sb.ToString();
    }
}
```

Vzorec kode 12: C# – objekt `XmlTag`, ki predstavlja značko XML in se ob klicu metode `ToString` vanjo serializira. Objekt `XmlAttribute` je definiran v Vzorcu kode 13.

```
public class XmlAttribute
{
    public string Name { get; set; }
    public object Value { get; set; }

    public XmlAttribute(string name)
    {
        Name = name.Trim();
    }
    public XmlAttribute(string name, object value) : this(name)
    {
        Value = value;
    }
```

Nadaljevanje na naslednji strani ...

Nadaljevanje s prejšnje strani ...

```
    public override string ToString()
    {
        char wrapper = Value is string && ((string)Value).Contains('"') ? ↩
            '\'' : '\"';
        return string.Format("{0}={1}", HttpUtility.HtmlEncode(Name), ↩
            HttpUtility.HtmlEncode(Value .ToString().Trim()).Wrap(wrapper));
    }
}
```

Vzorec kode 13: C# – objekt `XmlAttribute`, ki predstavlja atribut XML in se ob klicu metode `ToString` vanj serializira.

```
{
    "results" : [{
        "address_components" : [{
            "long_name" : "25",
            "short_name" : "25",
            "types" : [ "street_number" ]
        }, {
            "long_name" : "University of Ljubljana",
            "short_name" : "University of Ljubljana",
            "types" : [ "establishment" ]
        }, {
            "long_name" : "Tržaška cesta",
            "short_name" : "Tržaška cesta",
            "types" : [ "route" ]
        }, {
            "long_name" : "Ljubljana",
            "short_name" : "Ljubljana",
            "types" : [ "locality", "political" ]
        }, {
            "long_name" : "Slovenia",
            "short_name" : "SI",
            "types" : [ "country", "political" ]
        }, {
            "long_name" : "1000",
            "short_name" : "1000",
            "types" : [ "postal_code" ]
        }, {
            "long_name" : "Ljubljana",
            "short_name" : "Ljubljana",
            "types" : [ "postal_town" ]
        }],
```

Nadaljevanje na naslednji strani ...

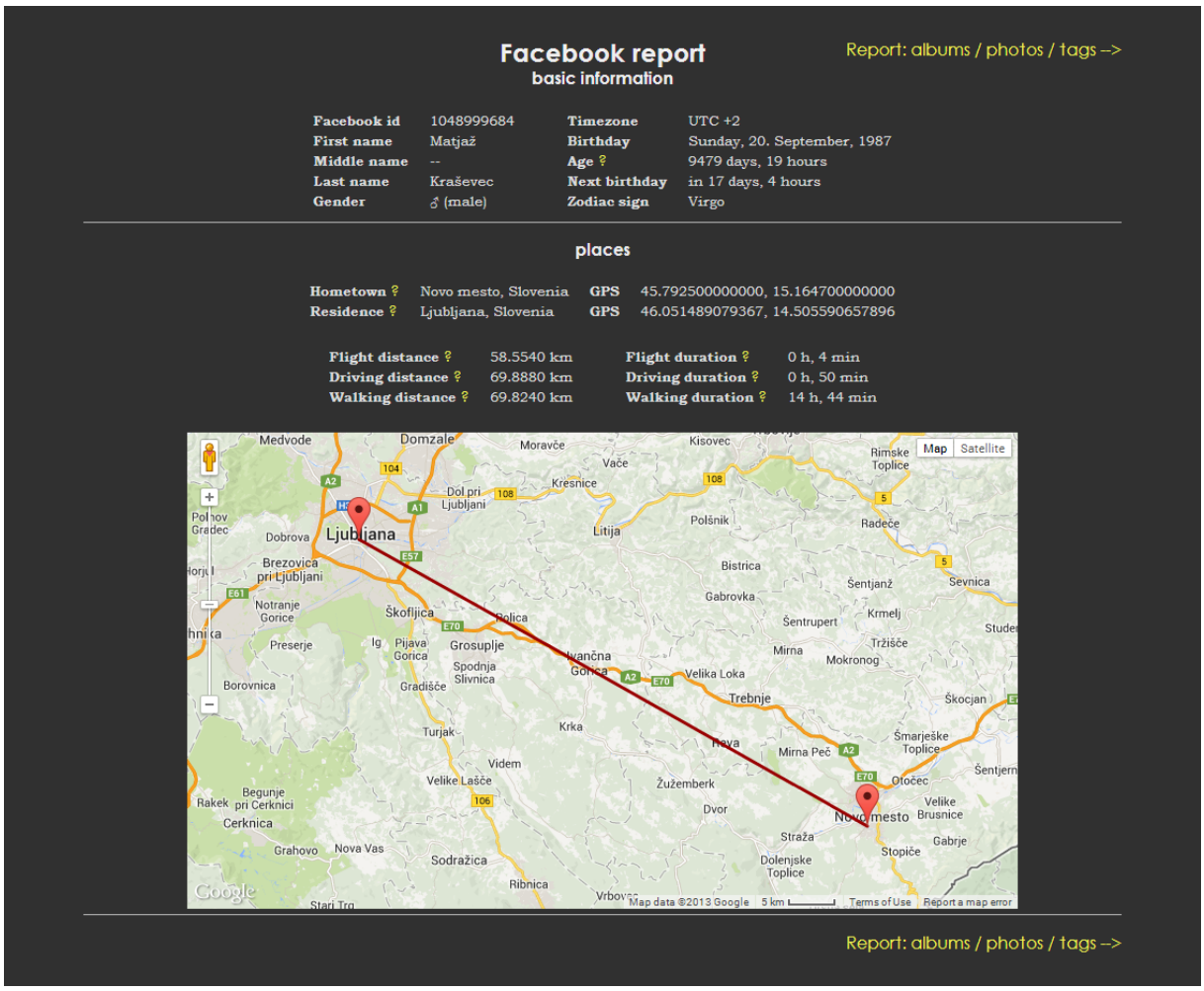
Nadaljevanje s prejšnje strani ...

```
        "formatted_address" : "Tržaška cesta 25, University of Ljubljana, ↩
           1000 Ljubljana, Slovenia",
        "geometry" : {
           "location" : {
              "lat" : 46.0448994,
              "lng" : 14.4892307
           },
           location_type: "ROOFTOP",
           viewport: {
              northeast: {
                 lat: 46.04624838029149,
                 lng: 14.4905796802915
              },
              southwest: {
                 lat: 46.0435504197085,
                 lng: 14.4878817197085
              }
           }
        },
        "types" : [ "street_address" ]
     },
     ⋮
     ],
     "status" : "OK"
}
```

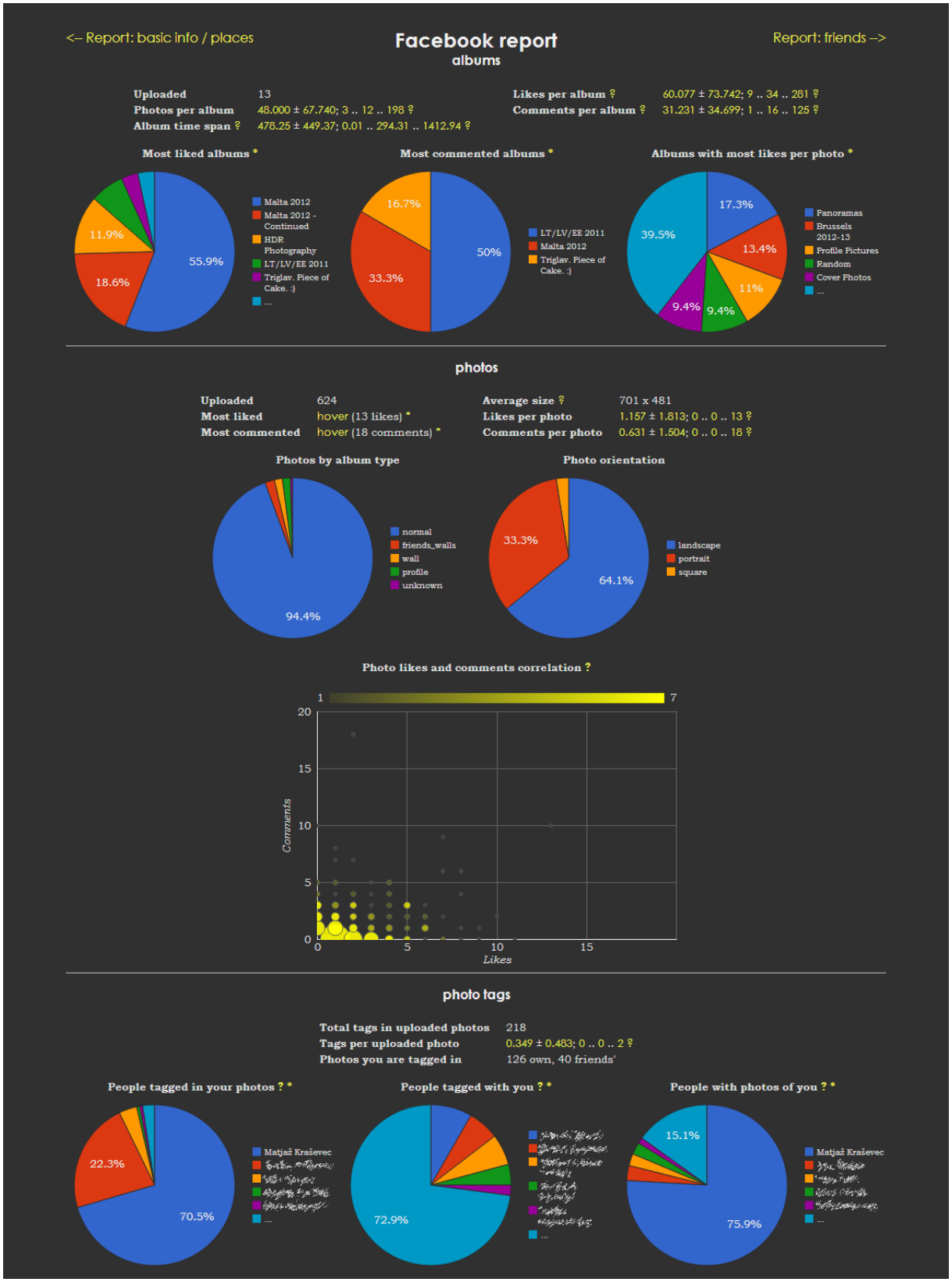
Vzorec kode 14: JSON – del odziva API-ja Google Maps Geocoding na zahtevo GET za obratno geokodiranje, razvidno iz Vzorca kode 3.4 v Poglavju 3.6.2). V polju `address_components` so navedeni podatki fizičnega naslova, vključno s hišno številko, imenom ustanove, ulico, mestom, poštno številko in državo.

# Posnetki zaslona Poročila

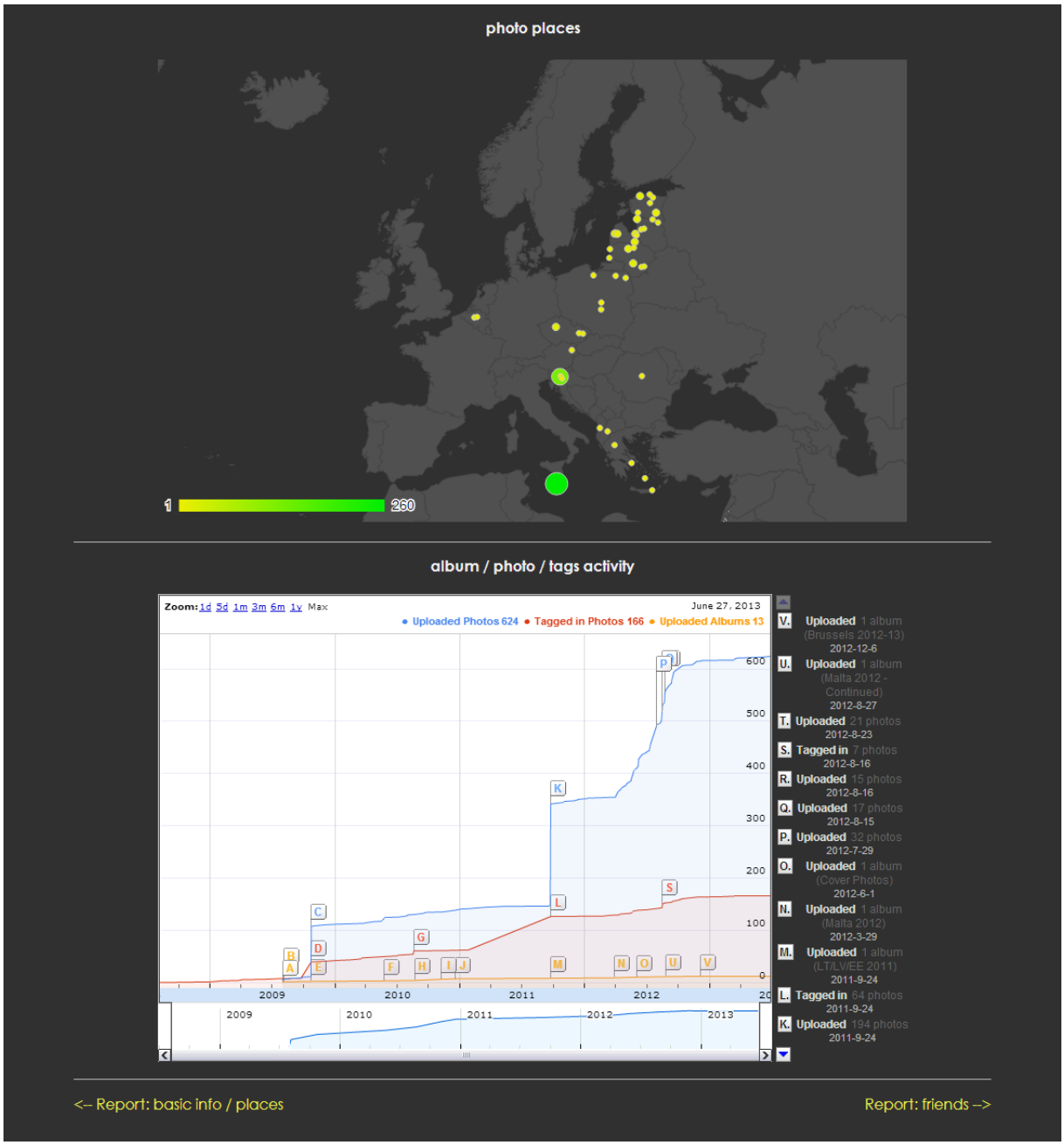
Sledijo slike s posnetki zaslona posameznih sekcij analitičnega poročila o podatkih s Facebooka, opisanega v Poglavju 3.6.3, ki si ga uporabniki lahko generirajo v aplikaciji *ObrazoKnjigoGled*. Zaradi varstva osebnih podatkov smo določene informacije v slikah zakrili.



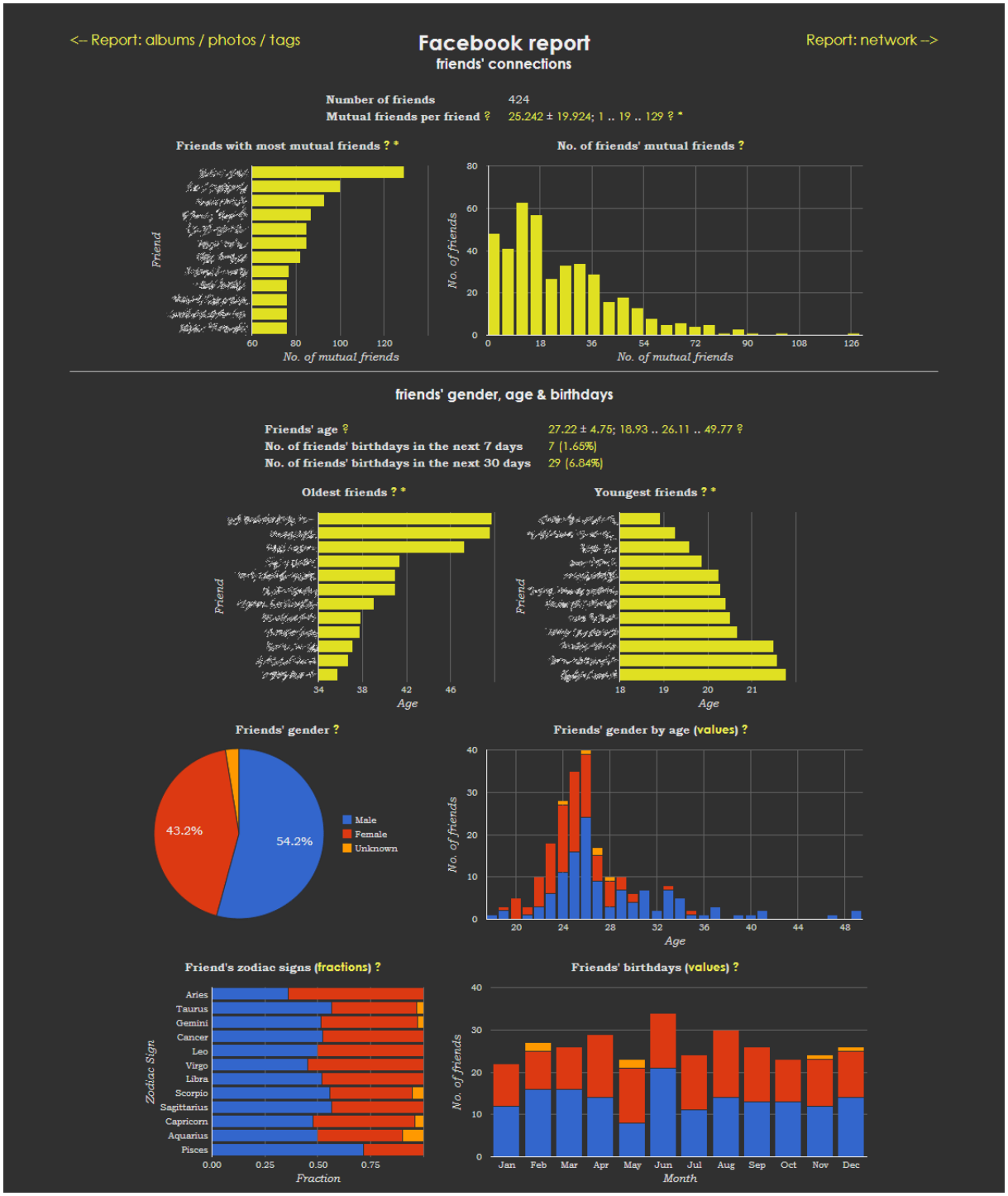

Slika 1: Sekcija o osnovnih in lokacijskih podatkih uporabnika.



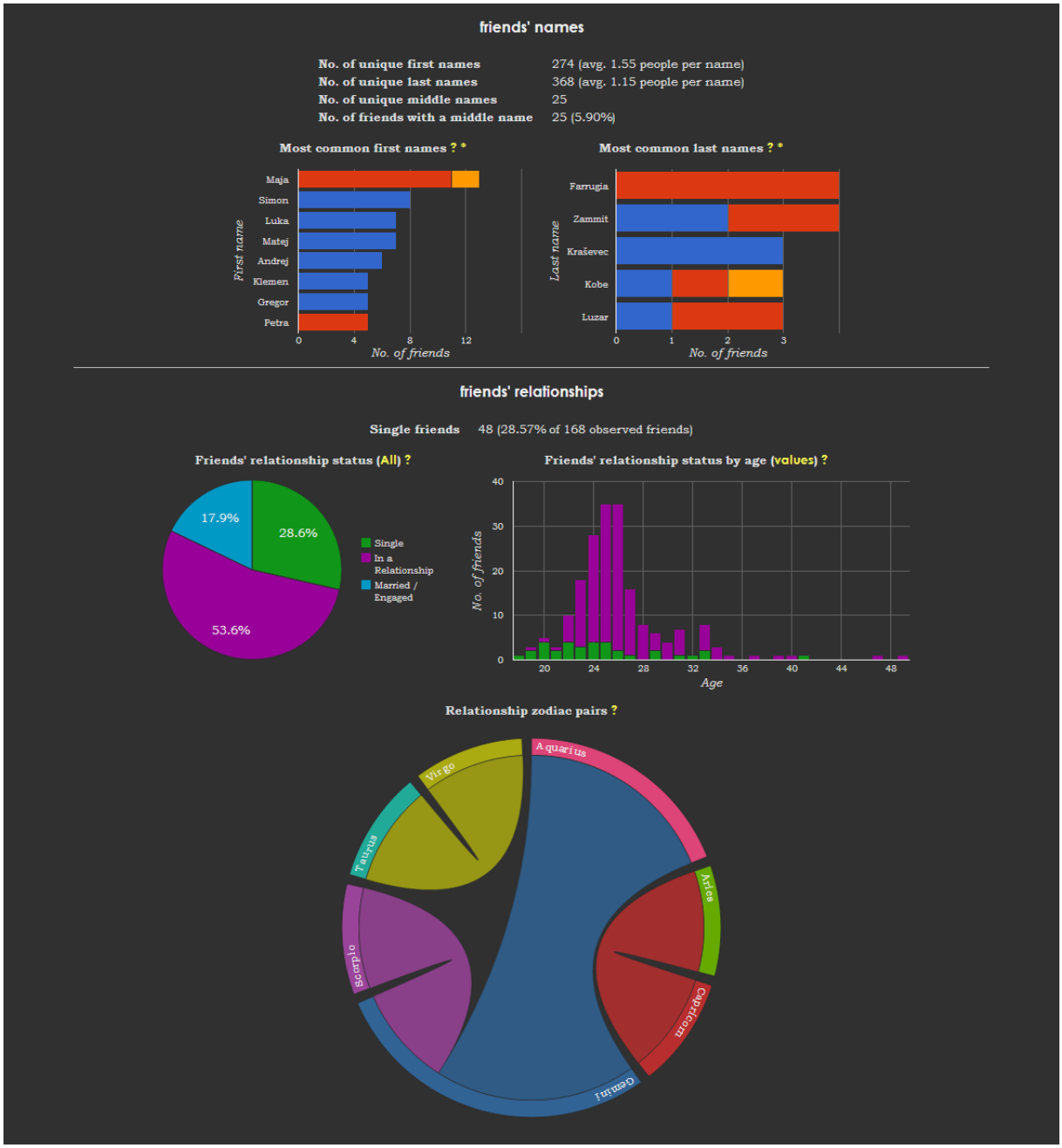

Slika 2: Sekcija o albumih, slikah in označbah (1/2).



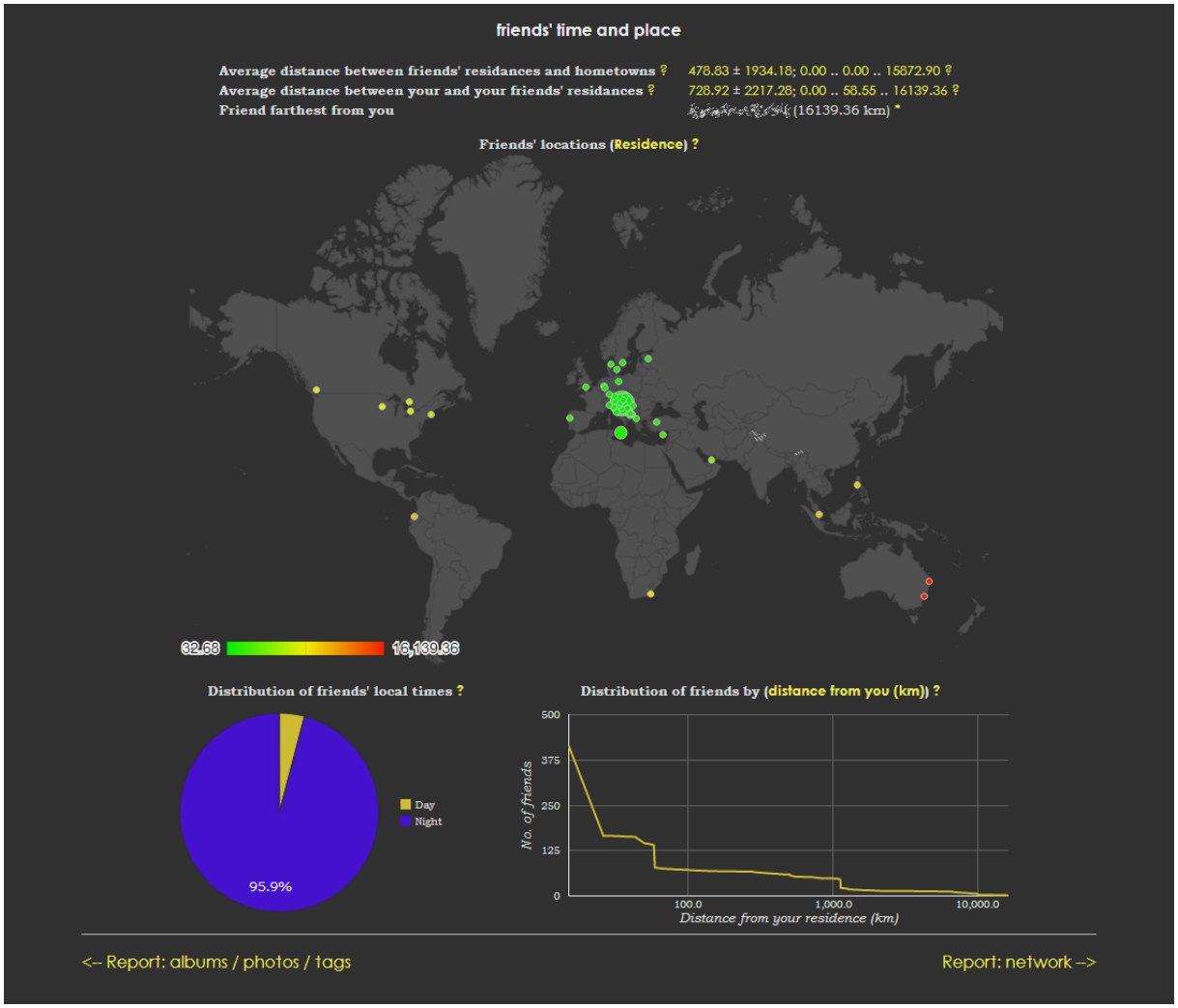

Slika 3: Sekcija o albumih, slikah in označbah (2/2).



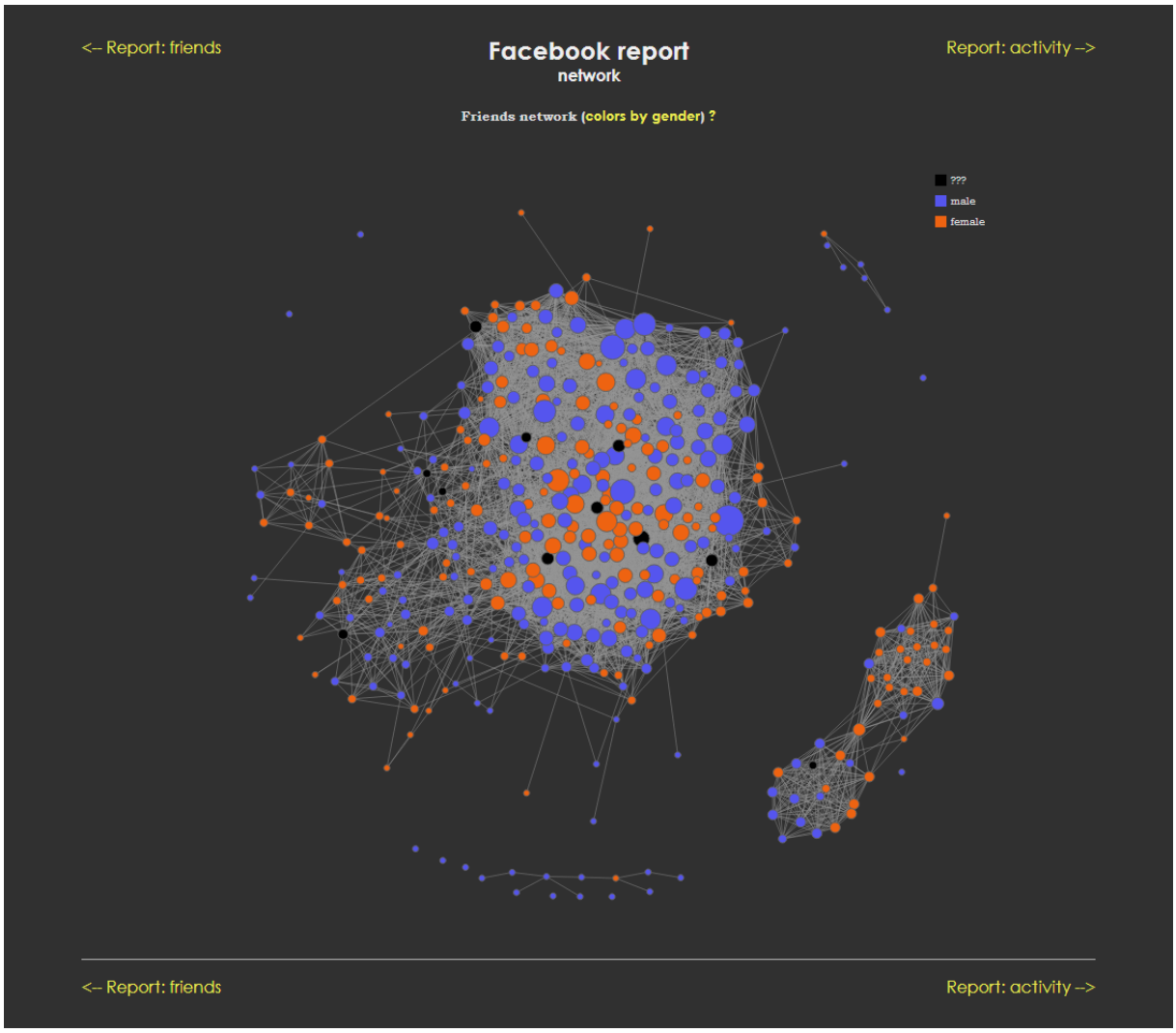

Slika 4: Sekcija o prijateljih (1/3).



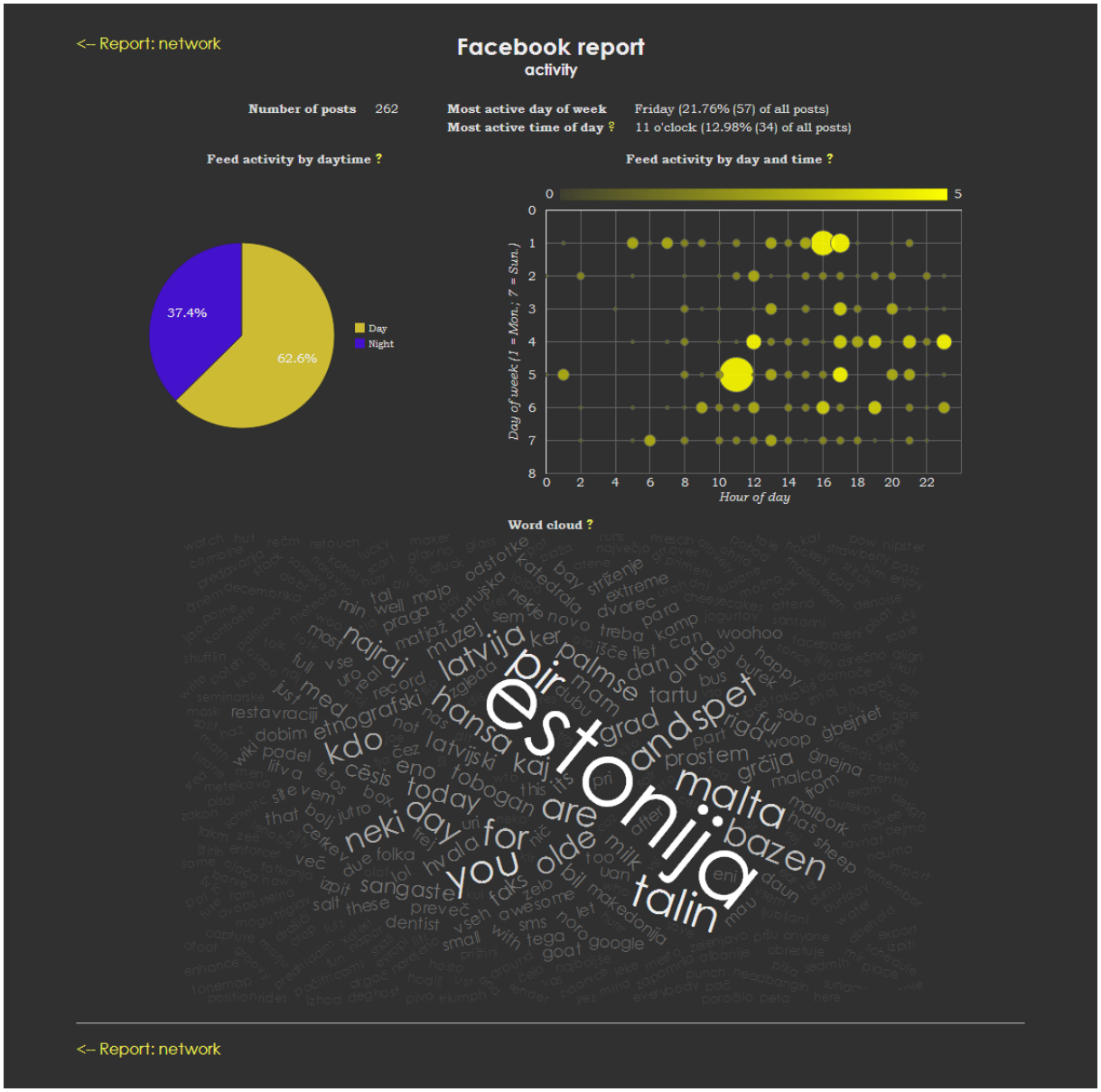

Slika 5: Sekcija o prijateljih (2/3).




Slika 6: Sekcija o prijateljih (3/3).




Slika 7: Sekcija o mreži prijateljev.




Slika 8: Sekcija o aktivnosti na Facebooku.