Masarykova univerzita
Fakulta informatiky

Design and implementation of a social network for making acquaintances

Bachelor thesis

Marek Bryša

Brno, spring 2012

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Marek Bryša

Advisor: doc. Ing. Michal Brandejs, CSc.

Acknowledgement

Thanks

abstract

Contents

Introduction

1 Design

In this chapter we will decide the design of a new social network for making acquaintances. Firstly we will research existing networks in this market segment. Next we will focus on user data protection aspects relevant in this context. Lastly we will evaluate pros and cons of those networks from both user experience and data protection point of view, and based on this we will make the design decision.

1.1 Existing social networks for making acquaintances

According to [12], there have been more than 844 lifestyle and dating sites in 2004. We shall focus on those most visited or gaining a lot of popularity.

1.1.1 PlentyofFish

PlentyofFish (`http://www.plentyoffish.com/`) was founded in 2003 in Canada. It generates most of it's revenue through advertising and some premium services. Unfortunately, it currently only serves users from Canada, UK, US, Australia, Ireland, New Zealand, Spain, France, Italy and Germany, so the author could not sign up at all.

From the publicly available information, it allows users to create a profile, search for others, message and chat with them. A *Chemistry test* and some other methods of finding a match are offered, but without explaining precisely how they work. [9]

1.1.2 Match.com

Match.com (`http://www.match.com`) was launched in 1995 and is one of the oldest networks. It requires a paid subscription ranging from 34.90 EUR for one month to 77.40 EUR for 6 months.

After signing up, the user is asked to upload a profile photo and fill in a detailed questionnaire about his or her character, interests, activities, relationships and preferences. Based on this information, the system tries to find the best matching partner. The user can then add the match to his or her favourites, follow their profile and message them. There is a special option to *wink* at them that can be used to quickly bring attention of the user and wait for their response to quickly assess their general interest without the need to send a message. [8]

1.1.3 OkCupid

OkCupid (`http://www.okcupid.com`) started in 2004. It claims to be the fastest growing website for making acquaintances. It is ad-supported and the essential features are free to use. A paid subscription called *A-list* is also available for 14.95 USD/month. It removes the ads, allows advaced searching, changing of user name etc.

Matches can be found through search using general criteria or by filling out questionnaires. A user can also create a his own questions, set their importance and expected answers. When another user fills them in, the system calculates a match percentage. This process is probably unique to OkCupid. [6]

1.1.4 eHarmony

eHarmony (`http://www.eharmony.com`) is a paid service that was launched in 2000. It claims to have more than 33 million members. Subscriptions cost from 59.95 USD for a month to 239.4 USD for 12 months. It is primarily focused on finding a partner for marriage.

The service uses personality tests, mathematical matching and expert advice to find the best match. There are separate subsites targeted for specific social groups such as Asians, Christians, Jews, gays, lesbians etc. A new user has to fill in a very detailed questionnaire about his current status, personality and preferences.[2]

1.2 User data protection

When using this kind of social networks, the user usually has to provide information about himself that is very sensitive, even intimate. Protection of this data is therefore a very serious concern.

The data is very valuable beyond it's original intent to find the best match. It can be used for instance to precisely target advertisements, give offers to buy new products and so on. Hence it is essential that the user is made clear how the information he enters on a website is used or if it is disclosed to third parties.

The user should also have the ability to choose what data is shared with other users. In the best case this control should be very fine, i.e. the user should not be forced to share information in blocks, should be able to deny specified users access to his profile or parts of his profile etc. There should also be a simple tool to preview one's profile in the way others can see it.

If a user deletes any data on his profile, it should be physically deleted from all the servers as well, unless it is expressly stated otherwise (e.g. for backup purposes).

Any changes to the privacy policy of a website should be only done with sufficient prior notice, and preferably be opt-in. In this context, the user must have the ability to simply download all his or her data in a package and delete the account.

The language of the privacy policy should be as simple as possible, for every user to clearly understand it. Almost no one will read a lengthy legal text which can lead to unfortunate misunderstandings later.

It goes almost without saying that the servers must be well protected from hacker attacks, especially when they contain this kind of sensitive data. A successful attack would not only harm the users, but probably mark the end for the website. Ideally there should be a regular security audit that the users can review.

1.3 The design decision

From the research of existing social networks for making acquaintances we can conclude the following points and issues:

- The target audience are single people from about 20 to 60 years old.
- Many require a paid subscription to access even the most basic functionality.
- All require new users to fill in a long, detailed and intimate questionnaire. This can discourage many users.

- Therefore all collect very sensitive user data that could be potentially misused.
- All offer a method to quickly find a matching partner, but then require an action from one of the users to make a first contact. Some users might have trouble finding courage to do so.

Finkel et al. (2012) [3] state more psychological problems with on-line dating sites. They also claim mathematical algorithms do not prove to be especially successful. To solve most of these issues, the author has come up with this idea for the new social network:

- The users will provide only general information: e-mail address, gender, year of birth, approximate location (county level), interest in men or women and a single profile photo.
- Based on simple search criteria such as age range, relative location to them (i.e. same county, neighbouring counties), they will browse profile pictures of other users one by one and mark the ones they like.
- Only once two users match their 'like' mark, both will be notified, added to their contact lists and be able to engage in real-time chat. Then they can get to know each other and possibly arrange a meeting.

This way only very little information is gathered in the database which brings the user data privacy problems to a minimum, and it is not necessary to fill in any lengthy questionnaires. Users need not be shy when marking people they like, because until the mark is matched, the other person will not know about it.

However this also brings some new issues. Because the marking of others is essentially only based on their looks, the target audience is going to be reduced to users for whom it is an important criteria. That means mostly younger people seeking fun rather than a serious relationship.

2 Implementation

Here we will focus on the implementation of the social network based on the basic design from the previous chapter. First we will research and compare technologies that can be used to achieve the goal, based on specific criteria. Then we will walk through interesting parts of the implementation from both users' and programmers' point of view. Source codes of the implementation are available on the attached CD, with some parts omitted for security reasons.

2.1 Technologies

It is very important to choose the right technologies for the implementation of a project. We need to find the most suitable HTML5 web application framework and a data store, if one is not hard-wired into the framework. With the exception of OkCupid which claims to use a proprietary C++ framework, existing social networks do not disclose technologies they use. The author has devised the following criteria for the evaluation of available technologies:

- **Availability for commercial use free of charge**
  Because of budgetary constrains, the technology must be free for commercial use. The project may later generate revenue through the use advertising.

- **General suitability for the project**
  It must facilitate creation of a website. It is expected that there will be a lot of HTTP requests that will make only little changes to the database, e.g. marking of photos a user likes. The data model will be quite simple. There must be an easy way of making HTTP push[1] communication to enable real-time chat.

- **Performance, scalability and stability**
  Again due to the low budget, the software must utilize hardware as efficiently as possible. The user base could potentially

1. "HTTP server push (also known as HTTP streaming) is a mechanism for sending data from a web server to a web browser." `http://en.wikipedia.org/wiki/HTTP_push`, 2012-04-08

grow very rapidly. It is therefore essential that all the system can match the growth cost-efficiently. The framework should have a good track record of runtime stability.

- **Ease of development and developer community size**
  It should be easy to implement the project and good documentation is welcome. The framework should have a sufficient community with which a developer can try to solve potential issues.

- **Codebase stability**
  The technologies should be past their rapid development phases and the core APIs should be stable. This minimizes the effort needed to transition the project to a newer version of the framework.

- **Innovation factor**
  Younger technologies are preferred as their use can lead to innovation and discovery of new approaches to problems.

Because of the first criterion, our interest shall only be in open source frameworks.

2.1.1 General suitability for the project

The author is skilled in JavaScript, PHP, Python and Ruby, so we will further examine frameworks based on those languages. All have been used for HTTP server programming for a long time, except for JavaScript that has emerged in recent years in the Node.js platform.

	Node.js	PHP	Python	Ruby
Simple	Express.js	plain PHP	CherryPy	Sinatra
Full MVC	Locomotive, Railway.js	Zend, CakePHP	Dajngo, web2py	Ruby on Rails

Table 2.1: Classification of web frameworks

Table 2.1 shows a basic classification of selected web frameworks by programming language and comprehensiveness of features they provide. Simple frameworks generally only provide a way to route HTTP requests to methods, parse HTTP headers and to send a response. Other features can be added on using plug-ins or modules. Full MVC[2] frameworks also have an ORM[3] engine for models and generate HTML views using a templating engine.

Because the project's uncomplicated data model would not utilize the complex feature set of full MVC frameworks and those could limit flexibility, we will further only focus on the simple ones, i.e. Express.js, plain PHP, CherryPy and Sinatra.

2.1.2 Performance, scalability and stability

Let us first compare performance of the languages and their virtual machines themselves. We can use results from *The Computer Language Benchmarks Game* [4]. It uses several algorithms written in different programming languages to measure their speed.

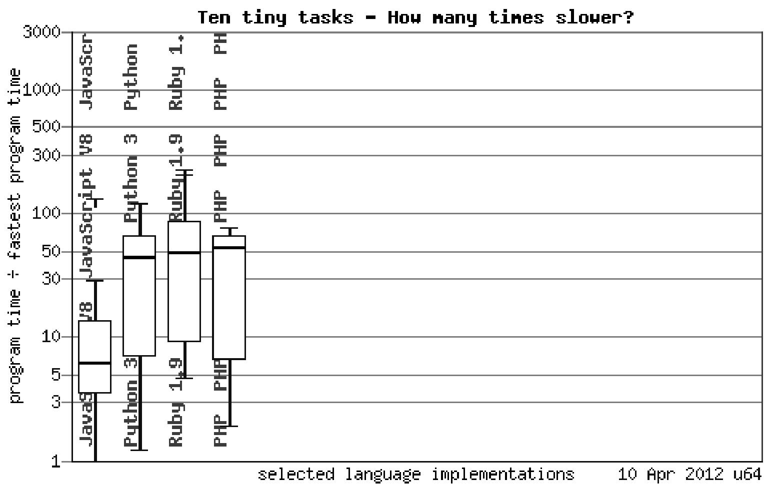

Figure 2.1: Language benchmark of V8 Engine, Python 3, Ruby 1.9 and PHP 5.4.0. Lower means faster. Source: [4]

A box plot of the benchmark is shown in figure 2.1. The vertical axis indicates how many times the language is slower than the

2. Model-View-Controller
3. Object-Relational Mapping

fastest one (currently Intel Fortran 12.1). Out of the languages under consideration, Node.js on average 6 times faster than the rest[4].

Next we will benchmark the performance of the HTTP handling of the frameworks. All the tests will be run on a dual-core i686 Linux 3.2.11 PC with 4GB of RAM. We will use the Apache HTTP server benchmarking tool (ab). Source codes for Sinatra and Node.js are in listings 1 and 2, the other two for PHP and CherryPy are on the attached CD.

Listing 1 Benchmark for Sinatra in Ruby

```
require 'sinatra'

get '/add' do
    (params[:a].to_i+params[:b].to_i).to_s
end

get '/sleep' do
    ms=params[:ms].to_i
    sleep(ms/1000.0)
    "Slept %s miliseconds." % ms
end
```

The add method has two parameters a, b and simply returns their sum. It's purpose is to simulate simple GET parameter parsing and response.

The sleep method has one parameter ms. Execution is suspended for ms milliseconds and a simple response is sent. This is intended to simulate a database query that takes given time. We will use a 20 ms delay.

Node.js uses its internal HTTP server, Sinatra uses the Thin server, CherryPy uses its internal WSGI server and PHP is hosted through mod_php on the Apache server. Unlike CherryPy and Apache that use a thread pool to serve requests, Node.js and Thin use the libevent that utilizes epoll on Linux and kqueue on FreeBDS theoretically allowing better concurrency. Node.js is also strictly single-threaded.

Here is a list of versions and parameters used:

- Node.js 0.6.13

4. Node.js uses the V8 Engine internally

Listing 2 Benchmark for Node.js in JavaScript

```
var app = require('express').createServer();
var util = require('util');

app.get('/add', function(req, res){
  var x=parseInt(req.param('a'))+parseInt(req.param('b'));
  res.send(x.toString());
});

app.get('/sleep', function(req, res){
  var ms=parseInt(req.param('ms'));
  setTimeout(function() {
    res.send(util.format('Slept %s miliseconds.', ms));
  }, ms);
});

app.listen(3000);
```

- PHP 5.3.10, Apache 2.2.22
- Python 3.2.2, CherryPy 3.2.2, 100 threads in the pool
- Ruby 1.9.3p125, Sinatra 1.2.7, Thin 1.3.3
- All logging including access is disabled.
- 5000 request per test
- Concurrency $\in \{1, 10, 30, 50, 100, 200, 300, 500, 700, 1000\}$
- `ab` is run 10 times for each parameter combination and a mean of successful requests per second is calculated.
- `/proc/sys/net/ipv4/tcp_tw_reuse` set to 1. This allows reuse of sockets in the `TIME_WAIT` state. This is a recommended setting for high concurrency web servers.
- 2 seconds of waiting time between each run of `ab`.
- In case one of the runs fails (i.e. any of the 5000 requests fails), a score of 0 request per second is awarded for the run.
- Source code of the Python script used to perform the benchmark can be found in attached file `bench.py`.

Graph 2.2 shows the results of the `add` benchmark. We can see that PHP keeps up with Node.js until the 100 concurrent requests mark, then declines sharply. Node.js is able to serve about 4700 request per second regardless of concurrency.

Graph 2.3 shows the results of the `sleep` benchmark. Node.js is again the clear winner with about 4700 request per second regardless of concurrency. This is because Node.js's `setTimeout`, as well as any database query, is non-blocking. Once the query is made, Node.js moves to serve other requests, until the query result is received.

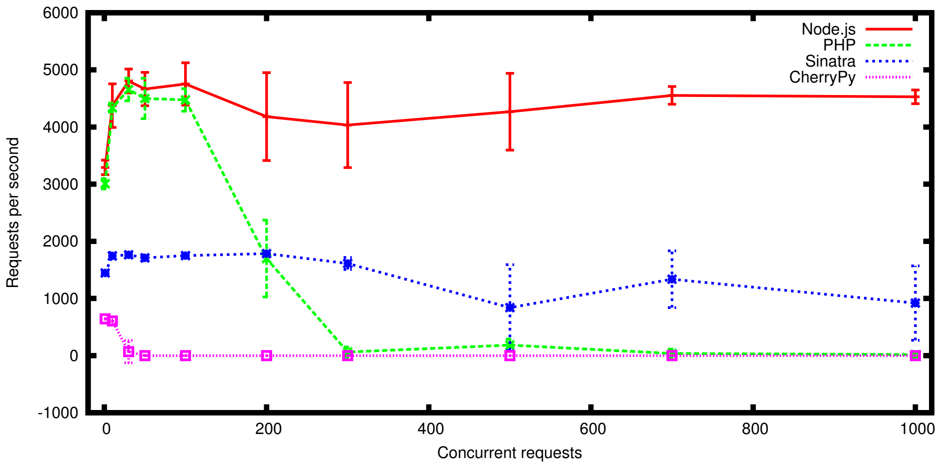

Figure 2.2: Benchmark of the `add` method. Error bars denote standard deviation.

Node.js and Sinatra on Thin are the only frameworks to remain stable with increasing load. PHP and CherryPy have started dropping requests at concurrency levels of 300 and 30 respectively.

In conclusion, Node.js performs the best as both a language and a HTTP server framework and remains stable under any load. Sinatra is also stable, however it is slower which could be partially solved by the use of clustering. Under low loads PHP on Apache is just as fast as Node.js, but it's stability is hardly acceptable. CherryPy is eliminated.

2.1.3 Ease of development and developer community size

All the frameworks in the comparison provide very similar levels of functionality. From the author's experience, Ruby permits the code to be shortest at the slight expense of readability. JavaScript on the other hand requires the longest code and can be a little tricky. Compare again listings 1 and 2.

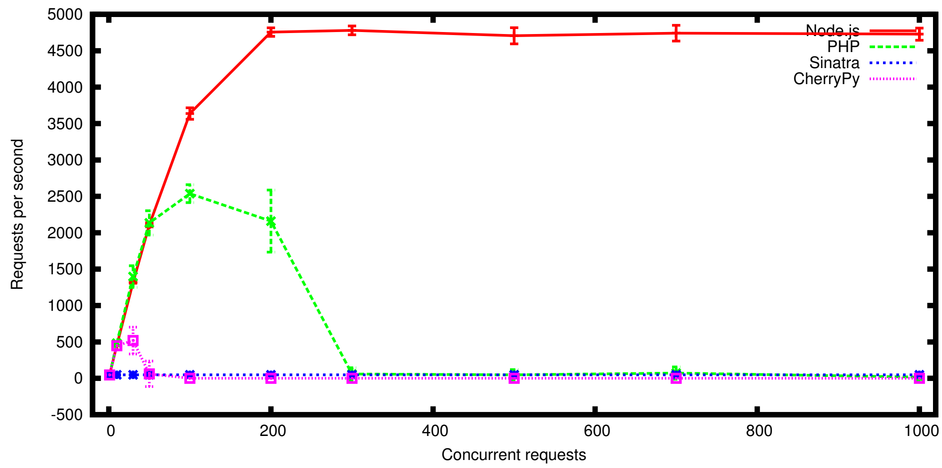

Figure 2.3: Benchmark of the `sleep` method. Error bars denote standard deviation.

PHP is arguably the most used framework for web programming, hence it has the biggest developer community. It is well documented in many languages. Countless modules, snippets and add-on libraries are available.

Node.js and Sinatra have a smaller but very active community. Thousands of modules and add-ons are available through their package managers such as npm for Node.js and RubyGems for Sinatra. Both are adequately documented if not deeply.

2.1.4 Codebase stability

PHP's codebase is the most stable. It is the oldest framework in the comparison, language and API changes are almost non-existent and don't break backwards compatibility.

Sinatra and Node.js split the second place. There have still been some changes to the Ruby language from version 1.8 to 1.9 that could break backwards compatibility. Node.js is still under active development, however the API tends to only grow and not change.

2.1.5 Innovation factor

Node.js is the clear winner in this criterion. It is the most innovative in that it uses JavaScript on the server side, event-based polling and non-blocking API.

Sinatra comes in the second place. It utilizes Ruby's advanced language constructions enabling tidy coding. PHP finishes last because of its age and lack of modern approaches.

2.1.6 The overall winner: Node.js

The author has chosen Node.js for the implementation of this project. It is just as suitable as other frameworks and clearly wins performance, scalability and stability tests. It is also the most innovative framework of recent years with growing developer community support.

There are two drawback to using Node.js. Firstly it is still relatively immature and bigger API changes could still happen. Secondly the use of JavaScript and non-blocking function calls can lead to poorly readable code. The author however believes that the advantages greatly outweigh this.

2.1.7 Data store

The decision to use a simple web framework gives us a free hand in choosing a separate data store. The traditional choice is an SQL database, e.g. MySQL and PostgreSQL. Lately NoSQL databases (MongoDB, CouchDB etc.) and advanced key-value storages such as Redis have come into focus. In a recent paper *Social-data storage-systems* [10] that compares all of above, no clear winner is given.

The author has chosen Redis. "Redis is an open source, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets and sorted sets."[1] Redis keeps the entire database in operating memory with optional regular persistent storage snapshots which allows very fast read/write access and good reliability. Benchmarks such as [11] show that Redis is about eight times as fast as MySQL when it comes to simple operations.

One of the drawbacks is that Redis mostly has simple commands so even moderately complicated operations are difficult to program.

The innovation factor is high because Redis is usually not used as a single storage for all the data. As a bonus, Redis contains a simple publisher-subscriber functionality which will come handy when implementing the real-time chat functionality.

2.1.8 Modules and libraries used

The following libraries and modules will be used to simplify implementation of the project:

Underscore.js "A utility-belt library for JavaScript that provides a lot of the functional programming support that you would expect in Prototype.js (or Ruby), but without extending any of the built-in JavaScript objects." [5] It will be used in both client and server JavaScript code.

Express.js A simple web framework on top of Node.js. Allows easy routing, GET and POST method parameter parsing and response sending. Uses the common `Connect` architecture, therefore is also a baseplate for other modules. `http://expressjs.com/`

node-jade A library for the Jade HTML templating engine. `http://jade-lang.com/`

redis A Node.js Redis client. Performance can be enhanced using the `hiredis` native backend. `https://github.com/visionmedia/connect-redis`

connect-redis A `Connect` module for saving of user session data to the Redis data store. Uses a signed cookie for client identification. `https://github.com/visionmedia/connect-redis`

node-sechash A library for calculation of cryptographically secure hashes to be used to store passwords. Automatically adds salt. `https://github.com/kbjr/node-sechash`

5. `http://documentcloud.github.com/underscore/` 2012-04-18

formaline An advanced HTTP POST request parser. Especially useful to handle file uploads. `https://github.com/rootslab/formaline`

node-gm A Node.js GraphicsMagick library. Facilitates image manipulation such as resizing, cropping and format conversion. `http://aheckmann.github.com/gm/`

i18n-node A simple internationalization library. `https://github.com/mashpie/i18n-node`

async A library to simplify asynchronous function calls on arrays of data, typically in series or parallel. `https://github.com/caolan/async`

RedBack "A fast, high-level Redis library for Node.JS that exposes an accessible and extensible interface to the Redis data types." [6] Its `RateLimit` class can be used for DDoS and spam prevention.

node-recaptcha A Node.js reCaptcha service client. Used for human verification. `https://github.com/mirhampt/node-recaptcha`

socket.io WebSocket and HTTP push library. `http://socket.io/`

node-amazon-ses A Node.js module for sending e-mails using the Amazon SES cloud service. `https://github.com/jjenkins/node-amazon-ses`

cluster A Node.js module for management of multiple server instances. `https://github.com/LearnBoost/cluster`

jQuery HTML DOM and CSS manipulation library for JavaScript. `http://jquery.com/`

jquery.validate jQuery plugin for HTML form validation. `http://bassistance.de/jquery-plugins/jquery-plugin-validation/`

jquery.Jcrop jQuery plugin for image cropping. `http://deepliquid.com/content/Jcrop.html`

6. `http://redbackjs.com/` 2012-04-18

jquery.elastic jQuery plugin that grows textareas automatically. `http://unwrongest.com/projects/elastic/`

UglifyJS A JavaScript minifier and obsfucator. `https://github.com/mishoo/UglifyJS`

2.2 Data model

Redis is a key-value storage, therefore there is no fixed schema for the database and no hierarchy per se. As a convention, the colon character is used to divide key names into logical subgroups, e.g. `user:1234:email`.

A complete plan of the structure of the database is included in appendix A.

2.2.1 Redis work-flow

Simple user parameters are stored as a Redis hash in order to reduce the total number of top-level keys which can have an impact on the performance. For example to access the e-mail address of user with `id` 1234, one has to execute Redis command

```
HGET user:1234 email
```

instead of

```
GET user:1234:email
```

An introduction to the data types and basic commands of Redis is available at `http://redis.io/topics/data-types`.

There is a way to search all key names for a given pattern using the `KEYS` command, but it is very slow compared to other commands. Therefore to keep track of the objects it the database, a set of their IDs must be kept. For instance to create a new user the sequence from listing 3 has to be executed.

First a user ID is generated by increasing the counter. Then a `multi` command is started. This guarantees that everything will be processed at once. Since Redis is single-threaded, it is atomic by default.

Also note the use of asynchronous function calls — one of the defining characteristics of Node.js.

Listing 3 An excerpt of user creation code

```
client.INCRBY('user:counters:id', Math.floor(Math.random()*50+1),
  function(err,replies) {
  var id = replies;
  var multi=client.multi();
  //...
  multi.HMSET('user:'+id, 'id', id, 'email', email,
    'sechash', sechash.strongHashSync('sha1', pass1, null, 5),
    'activated', false, 'vercode', vercode,
    'cwoeid', 0, 'swoeid', 0, 'owoeid', 0,
    'helpMode', true);
  multi.SADD('user:sets:email', email);
  multi.SADD('user:sets:id', id);
  multi.SADD('user:'+id+':visited', id);
  multi.HSET('user:hashes:email2id', email, id);
  //...
  multi.exec(function(err, replies) {
    //...
  });
});
```

2.3 Basic functionality

In this section we will cover interesting parts of the functionality that are more or less directly accessible for the user.

2.3.1 User registration

The user is presented with a registration form on the homepage. There are three fields: e-mail address, password and password confirmation. The e-mail field uses the HTML5 `email` type which causes supporting browsers to make the input easier, for instance by displaying the `@` character on a virtual keyboard on touch input devices. A minimum of 6 characters is required for the password. When the user clicks the Sign up button, the values are validated with the `jquery.validate` plugin and errors are highlighted, see figure 2.4. The pluing also prevents the form from being submitted with errors.

If there have been more than three registrations from a given IP address within the last 12 hours, a reCaptcha is displayed to prevent automated malicious user registrations. The `RateLimit` class of the

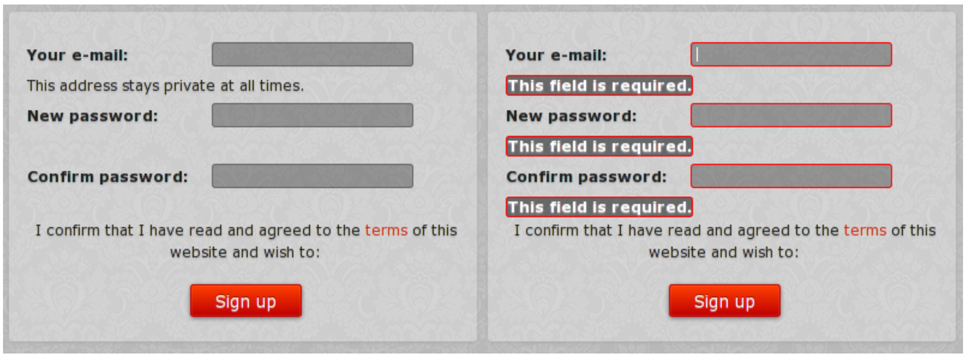

Figure 2.4: Sign up form.

RedBack module is used to efficiently store the number of registrations.

When the form is posted to `/signup`, all values and the reCaptcha are once again validated on the server side. This is indeed necessary, since it is very easy to circumvent any client side validation. User attributes are added to the database. The password is stored using the `sechash` module which makes it more difficult to guess users' passwords even if the database is hacked.

An e-mail is sent to the user's address, containing a welcome message and a verification code (for details of the implementation see section 2.4.1). Then the user is redirected to the second phase of the registration.

There the user is asked to enter his or her gender, year of birth and the verification code. It is a random number in the range $[100, 999)$. This is intended to make it simpler for the user to remember it for the few seconds while switching browser windows or tabs. On the other hand it would also enable malicious users to quickly guess the number, circumventing the e-mail verification. There is however a hard limit of 3 wrong attempts. An option to resend the verification code is also given, with a limit of 3 per 12 hours.

After that, the user is redirected to his or her profile page, where his or her approximate location must be entered and a profile photo uploaded in order to fully enable the account (see figure 2.6). More on geolocation in section 2.4.4.

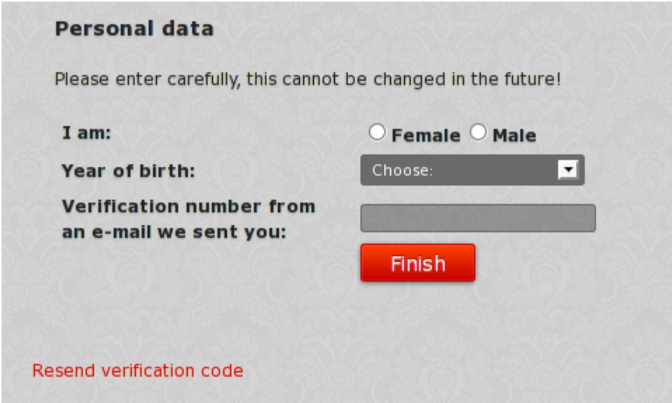

Figure 2.5: Second phase of signup.

Users have the ability to disable their accounts for the case they have found their acquaintance or are no longer looking for one. This causes notifications for their account to stop and they are no longer able to search for new users. Complete deletion of an account is also possible after entering the user's password and clicking on a link in a verification e-mail that is sent.

2.3.2 Photo upload and manipulation

The user can upload his or her profile photo through a `type="file"` field in a HTTP `POST` form using `enctype="multipart/form-data"` to `/me/newphoto`.

The `POST` data are passed to the `formaline` module. It makes sure that the file is no bigger than 4 MB, saves it to a temporary directory and runs the callback function on completion. The size limit is necessary primarily for security reasons, as processing of a bigger file could slow the server down.

Then a GraphicsMagick command is executed on the file:

```
gm identify -ping -format "%m %w %h" filename
```

It tries to identify the format, width and height of the image. The `-ping` option means that only the important parts of the file are read, ensuring quick execution. Only photos in `JPEG` or `PNG` format are passed through.

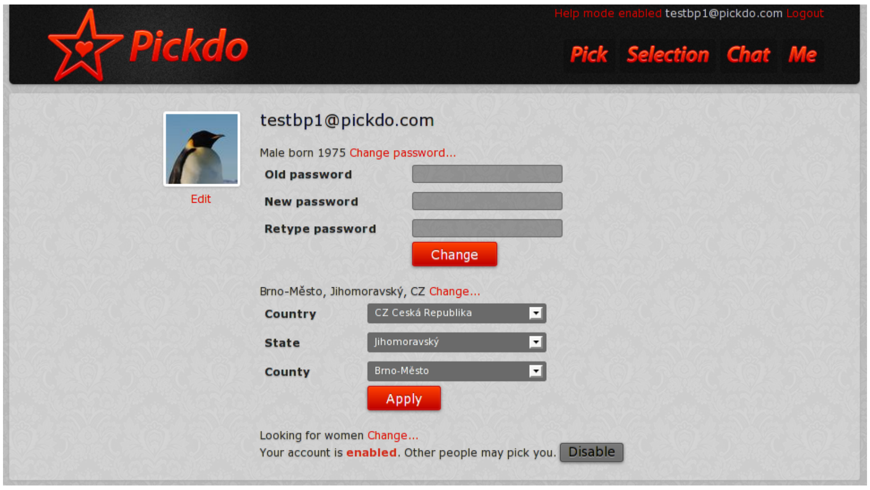

Figure 2.6: Profile and settigs page. (Author of the penguin photo: Hannes Grobe/AWI, CC-BY-SA 3.0)

The original file is then saved to a directory named in the following scheme:

`photos/ID-ID%10000/ID%10000/`

e.g. for a user with ID 23456 the directory will be `photos/20000/3456`. Because user IDs are increased by a random number in the range $[1, 50]$ with expected value 25, there will be approximately 400 folders in every 10000 level folder. This structure is necessary because most UNIX filesystems get slow when there are more than 1000 objects in a directory. In case the user base grows significantly and with it the potential of monetization, a commercial cloud based storage such as Amazon S3 would be a better choice.

After that, two `JPEG` thumbnails in maximum sizes 480×380 and 80×80 are created using GraphicsMagick, keeping the aspect and stripping any metadata, e.g. EXIF from a JPEG. Note that the `gm` command is always executed asynchronously in a separate process, so that it doesn't slow the HTTP server down.

The user also has the ability crop the uploaded picture directly on the website. A modified version of the `jquery.Jcrop` plugin facilitates the client UI and GraphicsMagick processing on the server side.

2.3.3 Acquaintance selection

The user has the ability to search for others using age and location criteria — gender can only be chosen in the profile and setting page as it stays constant for most users. The location can be one of these options: user's county, neighbouring counties, user's state, neighbouring states and user's country. Layout of the page is shown in figure 2.7.

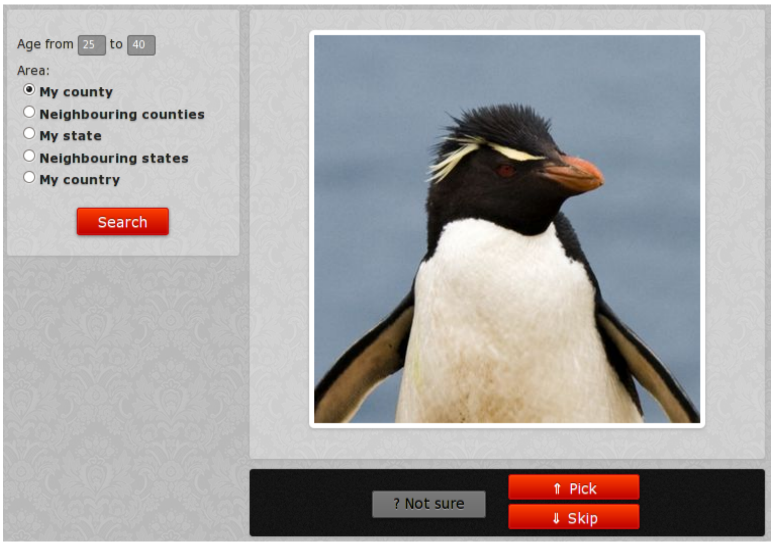

Figure 2.7: Search and selection screen. (Author of the penguin photo: Samuel Blanc, CC-BY-SA 3.0)

The form is asynchronously submitted to `/me/search` using HTTP `POST`. The parameters are validated on the server side and saved as a `JSON` string to the `user:<id> searchDefaults` hash. This is useful, because the client side JavaScript can use it to load the last used search parameters directly.

An excerpt of the first phase of the search procedure is shown in listing 4. First a union of sets of appropriate years of birth is created. It is saved in a temporary set that automatically expires in 10 seconds. Similarly a set of users is created according to the specified location.

These two sets, two other for gender selection and one with enabled users are then intersected and already seen users are subtracted, see listing 5. A maximum of 50 user IDs are returned as a `JSON` array.

The procedure can be in a simplified way written as this set fomula:

$$\left(\bigcup YOB_i \cap \bigcup GEO_i \cap GENDER \cap ENABLED\right) \setminus SEEN$$

We can see that this relatively simple task is rather complex to implement in Redis. The big upside however is that from testing with a database of 200 000 users, the whole HTTP request takes about 2 milliseconds on recent server hardware and all the command used have $O(n)$ or better complexity.

The user can then mark the resulting photos one-by-one, with more results loading automatically. The "Not sure" choice is used when a photo does not correctly depict the person. When such user then changes the photo, he or she is shown once again.

The user also has the ability to browse through visited photos and change the decision.

Listing 4 An excerpt of the search procedure — first phase

```
for(var i=yearTo;i<=yearFrom;i++) {
  yobSets.push('user:sets:yob:'+i);
}
client.INCR('tmp:counter', function(err1, yobUnionTmp) {
  var yobUnion='tmp:'+yobUnionTmp;
  var params1=[yobUnion];
  params1=params1.concat(yobSets);
  client.SUNIONSTORE(params1, function(err2, yobUnionCard) {
    client.EXPIRE(yobUnion,tmpExpire);
    var geoUnion='';
    if(area==1) {
      geoUnion='user:sets:geo:'+req.user.owoeid;
      searchAfterGeoUnion(req,res,geoUnion,yobUnion);
    } else if(area==2) {
      client.SMEMBERS('geo:sets:adjo:'+req.user.owoeid,
        function(err3, adjSet){
        if(adjSet.length>0) {
          client.INCR('tmp:counter', function(err4, geoUnionTmp) {
            geoUnion='tmp:'+geoUnionTmp;
            var params2=[geoUnion, 'user:sets:geo:'+req.user.owoeid];
            for(var adj in adjSet) {
              params2.push('user:sets:geo:'+adjSet[adj]);
            }
            client.SUNIONSTORE(params2, function(err, geoUnionCard) {
              client.EXPIRE(geoUnion,tmpExpire);
              searchAfterGeoUnion(req,res,geoUnion,yobUnion);
            });
          });
        } else {
          geoUnion='user:sets:geo:'+req.user.owoeid;
        }
      });
      //...
```

Listing 5 An excerpt of the search procedure — second phase

```
function searchAfterGeoUnion(req,res,geoUnion,yobUnion) {
  var tmpExpire=10;
  client.INCR('tmp:counter', function(err6, leftInterTmp) {
    var leftInter='tmp:'+leftInterTmp;
    client.SINTERSTORE(leftInter,
      'user:sets:gender:'+req.user.lf,
      'user:sets:lf:'+req.user.gender,
      'user:sets:enabled',
      geoUnion,
      yobUnion,
      function(err,leftInterCard) {
        client.EXPIRE(leftInter,tmpExpire);
        client.INCR('tmp:counter', function(err7, mainSetTmp) {
          var mainSet = 'tmp:'+mainSetTmp;
          client.SDIFFSTORE(
            mainSet,
            leftInter,
            'user:'+req.user.id+':visited',
            function(err8, mainSetCard) {

            client.EXPIRE(mainSet, tmpExpire);
            if(mainSetCard>50) {
              //limit to 50 and call searchAfterTotal
            } else {
              client.SMEMBERS(mainSet, function(err9, total) {
                searchAfterTotal(req,res,total);
//...
```

2.3.4 Notifications

Every time a user marks someone positively, a check is made to see if the liking is mutual. In that case both IDs are added to the respective `user:<ID1,ID2>:match_q` sorted sets (z-value[7] is current time) that act as queues. The z-value of the sorted set `user:zsets:match_pending` is increased by 1 for both users. Users that have disabled their accounts are no longer notified.

The `notification.js` script is running as a separate background process. It is used to regularly check for notifications for both matches and chat (more on that in section 2.3.5) and to send appropriate e-mails. It sets up a timer that ensures the match notifications are sent only once a day during the night. It can be gracefully quit by sending the `SIGQUIT` UNIX signal (in a similar way to the `worker-email.js` script shown later in listing 6).

The whole purpose of the rather complicated `matchNotify` function is to make sure as many as possible matches are made, but at most one new per user per day. This is the maximum bipartite matching problem. Efficient algorithms to correctly solve it exist. [5] However here, for the sake of simplicity, a naive algorithm is used — users with the lowest number of pending matches are handled first, with oldest pending matches taking precedence. This approach works reasonably well, as the vast majority of matches are one-on-one and even in case of slight congestion, a few days waiting time for the match to be handled is not a big problem. That said, this is something that could be improved.

When a match is handled, both users are notified using e-mail and are added to one another's contact lists to be able to engage in chat.

2.3.5 Chat

Implementation of the chat screen is shown in figure 2.8. Available contacts are listed on the left side. Those currently on-line are marked with a red square in the bottom right. If there are unread messages, their count is displayed in the top left. Red border marks the currently active chat session.

Message history is displayed in the centre and smooth-scrolls as

7. z is the value based on which the elements in the set are sorted.

new messages arrive. Textarea for message input is placed below. It automatically grows with the text length. If the box on the right of it is checked, the message is sent when the Return key is pressed, otherwise a new line is created. This allows both quick chatting without using the mouse and writing of properly formatted messages.

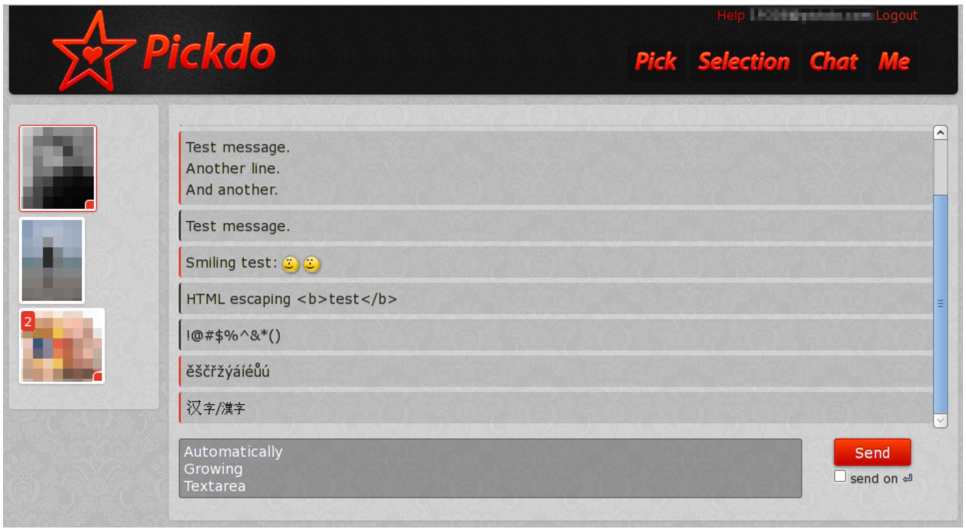

Figure 2.8: Chat page

A timer is set on the client side that makes an asynchronous request every 2.3 minutes to `/me/ping`. The server stores the last ping time in the `user:zsets:ping` sorted set and returns list a JSON list of contacts that are on-line, i.e. have sent their ping in the last 2.5 minutes.

All messages are asynchronously posted to `/me/sendmessage`. There it is stored as JSON object to the conversation history. If the number of messages in the history is above 100, the 10 oldest are deleted. Number of unread messages in the `user:<recipient ID>:unread <sender ID>` hash is increased. Also the `global:chat_q` sorted set is updated with current time as the z-value for the recipient ID. Lastly the message is published to both sender and recipient `push:chat:<ID>` Redis subscriber channels.

The `socketio.js` script is running as a separate background `socket.io` server on port 8080. All connections are authorized using the session cookie. Then it simply listens on the appropriate Redis `push:chat:<ID>` channels and pushes any new messages to the client. Currently the `xhr-polling` method is the only one enabled, since WebSocket has been disabled on most web browsers for security reasons and the protocol is going to have to be revised.

The `socket.io` client listens for incoming messages. When a new one is received, it is displayed in the chat history or a visual notification is made. When the message is read, this fact is acknowledged to `/me/ackmsg/<sender ID>`. There the appropriate unread message count is reset.

An e-mail notification about a new message is sent by the `notification.js` script if all of the following conditions are met (checked every minute):

- The message is unread and the user has not yet been notified about it.
- The user is off-line for a long time — more than 60 minutes have passed since the last ping.
- More than 3 minutes have passed since the last message addressed to the user.

This ensures the user is not overwhelmed with notifications.

2.4 Implementation in detail

In this section we will focus on aspects of the implementation that are more in the background, but are nonetheless essential for successful function of the project.

2.4.1 E-mail sending

Whenever the application needs to send an e-mail, it pushes the following JSON object to the `worker:email:{p1,p2}` Redis list, where `p2` has higher priority that `p2`.

```
{
  from:'Sender <sender@example.com>',
  to:['recipient1@example.com', 'recipient2@example.com'],
  subject:'Subject',
  body: {
    html:'E-mail body in <b>HTML</b> format.',
```

```
    text:'E-mail body in plan text format.',
}
```

The `worker-email.js` background process uses `async`'s `whilst` function and Redis's `BLPOP` command to non-blockingly poll for new e-mails as shown in listing 6. The `BLPOP` command pops an item from the list if the list is non-empty or waits for a specified amount of seconds for a new item. If it doesn't appear it returns `null`. The process is safely killable by sending the `SIGQUIT` UNIX signal.

Listing 6 The e-mail worker

```
async.whilst(function() {return !killed}, function(callback) {
  client.BLPOP('worker:email:p2', 'worker:email:p1', 10,
    function(err,replies) {
    if(replies) {
      ses.send(JSON.parse(replies[1]), function(data) {
        console.log(data);
        callback();
      });
    } else {
      callback();
    }
  });
}, function() {
  console.log('exit');
  process.exit();
});

process.on('SIGQUIT', function() {
  console.log('sigquit');
  killed=true;
});
```

The e-mails themselves are sent using the cloud Amazon Simple Email Service (SES – `http://aws.amazon.com/ses/`). It has favourable pricing at 0.10 USD per 1,000 e-mails plus data transfer at approx. 0.10 USD per 1 GB. Using this service spares the resources of running and administering one's own SMTP server and gives benefits in delivery rates, because Amazon checks outgoing e-mails for signs of spam, which gains it a more trusted status among e-mail client's anti-spam filters.

2.4.2 Security

Security of the application is essential for both users and operators. There are many attack vectors hackers could use to compromise them. [7] Here is a list of the attacks that apply here and measures that have been taken to prevent them:

Parameter tampering Because of the nature of HTML/HTTP, it is very easy to send parameters that are not allowed by the client side of the application. Hence, before any request is processed, it is passed through a series of `connect` filters. The `limit` filter built-in to `express` limits the size of `POST` parameters to a given amount — here 100 KB. `GET` parameters are automatically limited to 8KB by Node.js. Moreover lengths of all fields are then checked by the custom `paramLength` filter.

Cross-site scripting This attack is possible whenever user content is displayed to other users. In this project it happens in two places: the profile photo and chat. The profile photo is always checked to be of the correct format. All chat messages are properly HTML escaped before they are displayed.

SQL-injection Cannot be applied directly because Redis is a NoSQL data store. All Redis commands have a fixed structure and user content is always passed as binary-safe data and no plain text queries are used, as opposed to SQL. There is one exception: key names with `id` are constructed from user data (e.g. `"user:"+id+":email"`), but these parameters are always converted to integer before usage. Any such attacks are therefore impossible.

Authorization In the best case, all client-server communication should be run through a secure channel, i.e. SSL, TLS, HTTPS. This is however very demanding on the servers or requires a dedicated SSL acceleration hardware, both beign cost prohibitive for now.

Cookie theft Authorization is cookie-based, so the possibility of this attack indeed exists. However to minimize the risk, `connect` uses cryptographically signed cookies with fingerprints (e.g. client IP address, browser `User-Agent` header etc.), so the attacker would not only need to steal the cookie, but also to emulate all there parameters which is close to impossible.

2.4.3 Internationalization

Server-side internationalization is facilitated by the `i18n.js` module, which is a version of the original by Marcus Spiegel extended with this functionality:

- Option to set a default locale
- Get the locale from session automatically
- Translate messages to a locale specified by parameters independently of the request (the `l__` function)

Messages are stored as a JSON object in files, e.g. `locale/en.js`.

Because many user actions happen asychronously, client-side internationalization is also present. Messages are again stored as a JSON object (`locale/en_client.js`) and are directly inserted in the HTML file in a `<script>` tag creating the `t` global variable. There are no parametrized messages on the client side, so this simple approach suffices. Otherwise the server-side `i18n.js` module could be easily ported to client side, since both employ JavaScript.

2.4.4 Geolocation

All geolocation data used in the application comes from the Yahoo! GeoPlanet project. [13] It provides both a web service, which allows for searching of the database, and the source data tables. Here the `places` and `adjacencies` tables are used. Every place in the database has its unique constant ID called WOEID (Where On Earth ID).

The `places` table contains all the places available, their names, WOEIDs, types and parents' WOEIDs. The `adjacencies` table simply contains pairs of WOEIDs of the same level that are geographically adjacent.

The `geo_fill.py` Python script is used to fill the geolocation database of the application from the provided GeoPlanet database. Complete structure of the generated database is listed in appendix A under the `geo:` prefix.

2.4.5 Performance tuning

Both Node.js and Redis are single-threaded, therefore performance can be significantly improved by running multiple instances of them. With Node.js this can be easily achieved by using the `cluster` module that can manage starting, zero-downtime reloading and stopping of the instances.

With Redis this is more complicated, because the instances have their own data. Technique called *sharding* can be used to solve this. It spread keys with variable name such as `user:<ID>:email` across n instances by caclutating `ID` mod n. This is however impractical here, because set operations are heavily used and those cannot be done when the sets are stored in different instances. So at least the session data is stored in an instance separate from the rest. Redis also features simple master-slave replication that could also be employed.

All client-side JavaScript is combined into one file and minified using `uglify-js` to improve loading speed. For the same reason Gzip compression is utilized to serve text-based static files.

2.4.6 Graphical design and user experience

Attractive graphical design is essential for the success of any web application. The author has tried to do his best in this aspect. Advanced CSS3 effects such as gradients, text- and box-shadows, opacity and transitions are used as well as JavaScript animations.

Once the user logs in, all operations are done asynchronously. Not only does this reduce server load but also greatly improves user experience and responsiveness.

To help the user out while familiarizing with the website, *help mode* can be enabled. It then displays help bubbles when the user hover mouse over UI elements.

3 Conclusion

A Structure of the storage

`<id>` — mutable parameter
`*` — stored as hash

```
user:
  sets:
    email - registered e-mail addresses
    id
    enabled
    geo:<state/county/country woeid>
    gender:0
    gender:1
    lf:0
    lf:1
    yob:<yob>
  hashes:
    email2id
  counters:
    id
  *<id>:
    email
    sechash
    activated
    gender 1-female 0-male
    yob
    vercode
    cwoeid - country
    swoeid - state
    owoeid - county
    enabled - boolean
    userPhoto json:
      cropped - boolean
      cropCoords - [x1,y1,x2,y2]
      uploaded
    searchDefaults json:
      ageFrom
      ageTo
      area 1-county 2-neigh counties 3-state 4-n states 5-country
    locale
    helpMode boolean
    refid
  <id>:yes
  <id>:no
  <id>:maybe
```

```
  <id>:visited !cached union yes,no,maybe,me
  <id>:yes_t ZSET time
  <id>:no_t
  <id>:maybe_t
  <id>:yesFrom
  <id>:noFrom
  <id>:maybeFrom
  <id>:match ZSET time userId
  <id>:match_q ZSET time userId
  <id>:ref_count
  <id>:unread:*<id> unread count from
  zsets:
    ping time id
    match_pending count id
    photo_change time id
global:
  chat:<id1>:<id2> id1<id2 ZSET t message-json:
    from,to,body,time,seq
  chat_q ZSET time userId
  email_q ZSET time userId
  limit:
    signup - redback rateLimit
worker:
  email:
    p1 list - ses json
    p2
log:
  signup ZSET t  json id,ip,email
geo:
  sets:
    countries - set woeids
    states
    counties
    adjo:<county woeid> - adj counties
    adjs:<state woeid> - adj states
    child:<country,state woied> - child states,counties
  json:
    countries - json (name,woeid)
    states:<country woeid> - json (name,woeid)
    counties:*<state woeid> - json (name,woeid)
    name:<county woeid> - json (oname,sname,cname)
    oparent:*<owoeid> - county parent json [cwoeid,swoeid]
```

Bibliography

[1] Citrusbyte. Redis. `http://redis.io/`, 2012. [Online; accessed 10-April-2012].

[2] eHarmony, Inc. Why eHarmony? `http://www.eharmony.com/why/`, 2012. [Online; accessed 31-March-2012].

[3] Eli J. Finkel, Paul W. Eastwick, Benjamin R. Karney, Harry T. Reis, and Susan Sprecher. Online dating. *Psychological Science in the Public Interest*, 13(1):3–66, 2012.

[4] Brent Fulgham. The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`, 2012. [Online; accessed 10-April-2012].

[5] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, March 1986.

[6] Humor Rainbow, Inc. Okcupid. `http://www.okcupid.com`, 2012. [Online; accessed 27-March-2012].

[7] IBM Corporation. The dirty dozen: preventing common application-level hack attacks. `ftp://ftp.software.ibm.com/software/rational/web/whitepapers/r_wp_dirtydozen.pdf`, 2007. [Online; accessed 10-April-2012].

[8] Match.com, L.L.C. Match.com. `http://www.match.com`, 2012. [Online; accessed 27-March-2012].

[9] Plentyoffish Media, Inc. Plenty of FAQ. `http://www.pof.com/faq.aspx`, 2012. [Online; accessed 27-March-2012].

[10] Nicolas Ruflin, Helmar Burkhart, and Sven Rizzotti. Social-data storage-systems. In *Databases and Social Networks*, DBSocial '11, pages 7–12, New York, NY, USA, 2011. ACM.

[11] Ruturaj. Redis, Memcached, Tokyo Tyrant and MySQL comparision. `http://www.ruturaj.net/redis-memcached-tokyo-tyrant-and-mysql-comparision/`, 2009. [Online; accessed 10-April-2012].

[12] Wikipedia. Online dating service — wikipedia, the free encyclopedia, 2012. [Online; accessed 27-March-2012].

[13] Yahoo Inc. Yahoo GeoPlanet. `http://developer.yahoo.com/geo/geoplanet/data/`, 2012. [Online; accessed 10-April-2012].