

Université libre de Bruxelles

Advanced Databases Project
Developing of a social network using CouchDB

Aldar Saranov, Najim Essakali

Aldar.Saranov@ulb.ac.be

Najim.Essakali@ulb.ac.be

INFO-H-415 Advanced Databases (M-INFOS/F277)

Esteban Zimányi

December 2016

Project introduction

We decided to develop a simple social network for user interacting via Internet connection. The social networks became one of the most trending areas in World Wide Web since Web 2.0 rise in 2005 [1]. Developing of a social network definitely requires establishing of a database capable of storing enormous amount of data (posts, messages, media etc.) and operatively respond to user requests.

In order to remain competitive over the social network market a social network is forced to have high technical characteristics and withstand benchmarking with the others. We can determine following general technical demand for a modern social network by decreasing priorities:

1. Security of personal data.
2. Fast response time.
3. Scalability.
4. Data consistency.
5. Simple modifying of functionality.

CouchDB

Why CouchDB?

CAP

We may use CAP theorem to illustrate the initial set of databases which were the candidates for the database to be used in the social network developing. CAP theorem is based on choosing 2 of 3 possible conceptions:

1. Consistency - All nodes see the same data at the same time.
2. Availability - Every request gets a response on success/failure.
3. Partition Tolerance - System continues to work despite message loss or partial failure.

Due to the order of priorities we have specified in the introduction we can infer that most appropriate category in our case is AP.

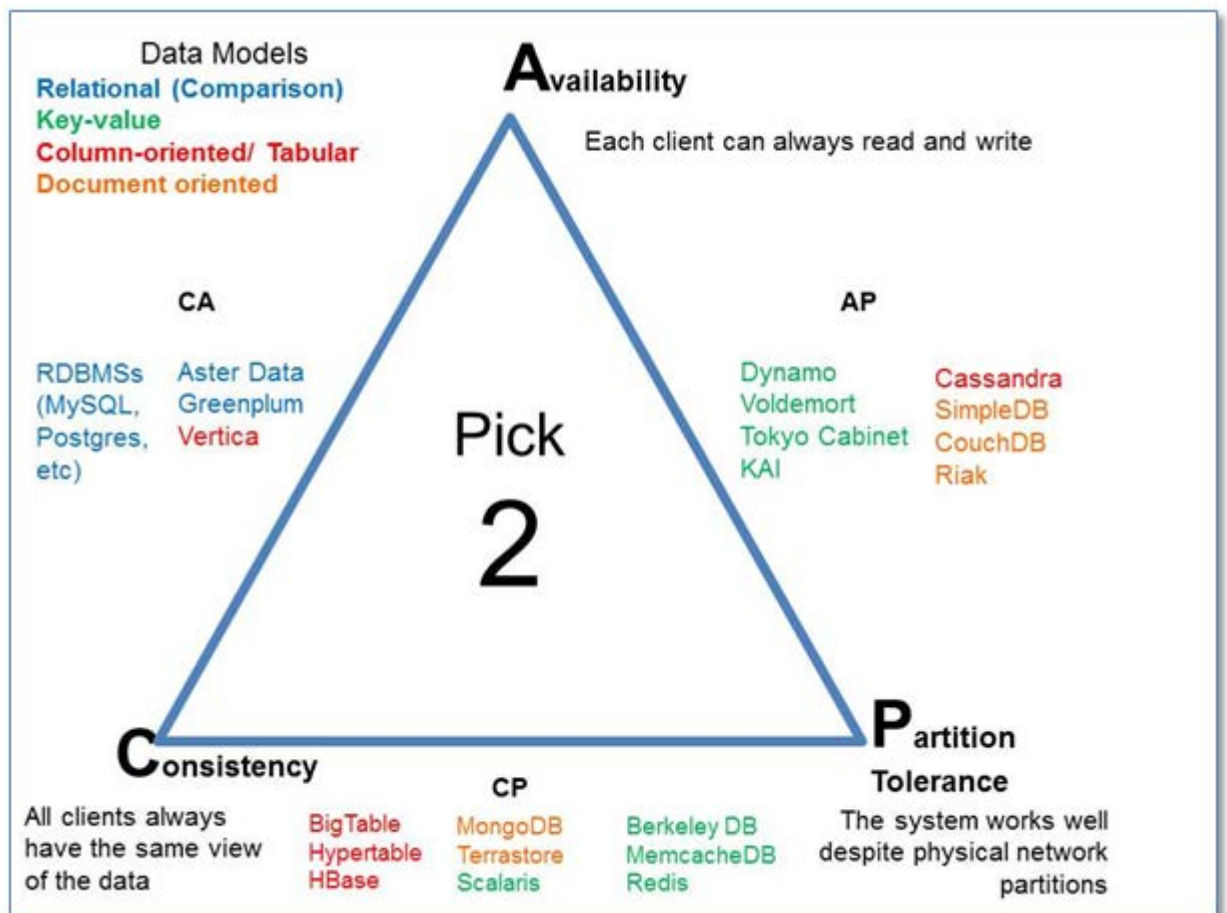


Figure 1. CAP theorem illustration.

Documented vs. key-value databases

The choice between key-value and document databases comes down to your data and application needs. If you usually retrieve data by key or ID value and don't need to support complex queries, a key-value database is a good option. If you don't need search capabilities beyond key lookup, a key-value database that supports searching may be sufficient. In our case where the model of the social network may include several sophisticatedly related entities: users, groups, messages, relationships and so on.

CouchDB is a document-oriented schemaless opensource database developed by Apache Software Foundation. CouchDB is a part of NoSQL conception databases and key-value databases in particular. CouchDB is accessible using a RESTful JavaScript Object Notation (JSON) API. The term "Couch" is an acronym for "Cluster Of Unreliable Commodity Hardware," reflecting the goal of CouchDB being extremely scalable, offering high availability and reliability, even while running on hardware that is typically

prone to failure. CouchDB was originally written in C++, but in April 2008, the project moved to the Erlang OTP platform for its emphasis on fault tolerance.

Relational databases define a strict structure and provide a rigid way to maintain data for a software application. On contrary in CouchDB case the term schemaless means that the database does not oblige the documents/objects to correspond to a certain logical form (besides the document formatting) while ensuring only a minor set of constraints (e.g. unique id). Most of the constraints should be implemented at application level what provides a high model flexibility to the developer.

CouchDB vs. SimpleDB

CouchDB was chosen over the SimpleDB because it adheres to the REST architecture therefore:

1. Uses classic GET, POST, PUT, DELETE requests.
2. Json allows to reduce the transmission data size.
3. Object indexing is directly under control of the user.

CouchDB vs. Riak

CouchDB was chosen over the Riak because it supports a much larger set of platforms (Linux, Windows, OS X, Solaris, Android, ...). In the rest of the benchmark parameters it mostly matches Riak.

Fast response time

To ensure low response time CouchDB uses B+ trees as data structure what assures that all essential operations like accessing/adding/deleting will be handled in logarithmic time. While accessing nodes with height less than 10, CouchDB can still index millions of elements.

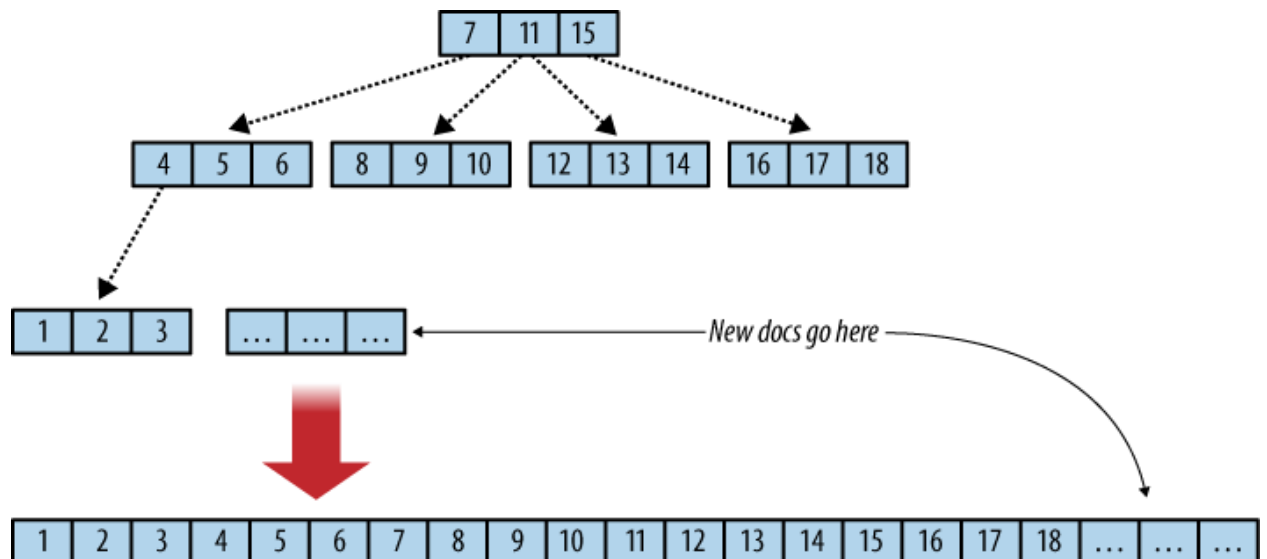


Figure 2. B-tree and append only.

Security of personal data

CouchDB has a quite simple security policy. There is a set of users (Admins) which are authorized to do any modifications to the database. This is called an “Admin party” mechanism. By default CouchDB server is only listening to the requests coming from 127.0.0.1 (localhost). List of operations which an admin can carry out:

1. Creating a database (PUT /database)
2. Deleting a database (DELETE /database)
3. Creating a design document (PUT /database/_design/app)
4. Updating a design document (PUT /database/_design/app?rev=1-4E2)
5. Deleting a design document (DELETE /database/_design/app?rev=1-6A7)
6. Triggering compaction (POST /_compact)
7. Reading the task status list (GET /_active_tasks)
8. Restarting the server (POST /_restart)
9. Reading the active configuration (GET /_config)
10. Updating the active configuration (PUT /_config)

CouchDB also provides built-in tools for password hashing what is especially useful in our case.

Persistence of data

CouchDB is fault-tolerant since there are many mechanisms which are designed to ensure the persistence of the stored data.

Operation consistency

However CouchDB B-tree implementation is slightly different from the canonic one. While it maintains all of the important properties, it adds Multi-Version Concurrency Control (MVCC) and an append-only design. B-trees are used to store the main database file as well as view indexes. One database is one B-tree, and one view index is one B-tree.

MVCC is designed to conduct concurrent read and write operations without using the locking of the system. Writes are serialized, allowing only one write operation at any point in time for any single database of the server. Write operations do not block read operations, thus there can be any number of read operations at any time. Each read operation represents a consistent view of the database. How this is accomplished is at the core of CouchDB model of storage.

The short answer is that because CouchDB uses append-only files, the B-tree root node must be rewritten every time the file is updated. However, old portions of the file will never change, so every old B-tree root, should you happen to have a pointer to it, will also point to a consistent snapshot of the database.

In a B-tree, data is being stored only in leaf nodes. CouchDB appends data only to the database file that keeps the B-tree on disk and grows only at the end. Both adding and deleting the documents are being recorded at the end of the file. The consequence is a robust database file. Computers fail for plenty of reasons, such as power loss or failing hardware. Since CouchDB does not overwrite any existing data, it cannot corrupt anything that has been written and committed to disk already.

If during in this process the power is cut-off occurs and CouchDB is being restarted later - the database file is in a consistent state and does not need a checkup. CouchDB starts to read the database file backwards. When it passes a footer pair, it is checking its state: if the first 2 kb are corrupted (figured out using checksum), CouchDB replaces it with the second footer and all is well. If the second footer is corrupt, CouchDB copies the first 2 kb over and all is well again. Only when both footers are flushed to disk successfully CouchDB will confirm that a write operation was successful. Data is never lost, and disk data is never corrupted.

Replication

CouchDB has a very powerful replication system. Replication synchronizes two copies of the same database, hence allowing users to have lower latency access to data. These databases can be situated on the same server or on two different server - CouchDB does not make any distinction. If changes take place in one copy of the database, replication will send these changes to the other copy.

Replication is a one-off operation: you send an HTTP request to CouchDB that includes a source and a target database, and CouchDB will send the changes from the source to the target. Replication request is called by making a POST request:

Listing 1. POST replicate request.

```
POST /_replicate HTTP/1.1
{"source": "database", "target": "http://example.org/database"} -H
"Content-Type: application/json"
```

When an admin ask CouchDB to replicate one database to another, it will go and compare the two databases to find out which documents on the source differ from the target and then submit a batch of the changed documents to the target until all changes are transferred. Changes include new documents, changed documents, and deleted documents. Documents that already exist on the target in the same revision are not transferred; only newer revisions are.

Such simple replication system can be useful for creating back-ups or snapshots for ensuring data persistence in our social network.

Alterability

Since CouchDB is a schema-less database it allows to conduct almost unlimited changing of the entity-relationship model. None of the documents has to have a certain set of attributes. If the social network developers that it is desired to implement new features for it then it can be done without pain for the already accumulated data. It is allowed to add any new attributes to the new documents. The only thing developers should care about is to implement correct data processing at application-level.

Documents in CouchDB allow attaching any MIME (Multipurpose Internet Mail Extensions) data type to the documents. Most common MIME types are audio, images,

text, video, public keys, 3D models and so on. This can help the developers to establish transferring of data of any useful content which makes sense in the terms of the developing social network.

General description

Main feature of CouchDB which differs it from most of other databases is that it uses HTTP as interface access protocol. So it makes technically possible to create real useful databases which could be directly accessed by client browsers. However such approach may be non-comfortable for a common user using simple browser to access data. At the same time most used configuration requires establishing of a web-server which handles the most of the business-logic of the system.

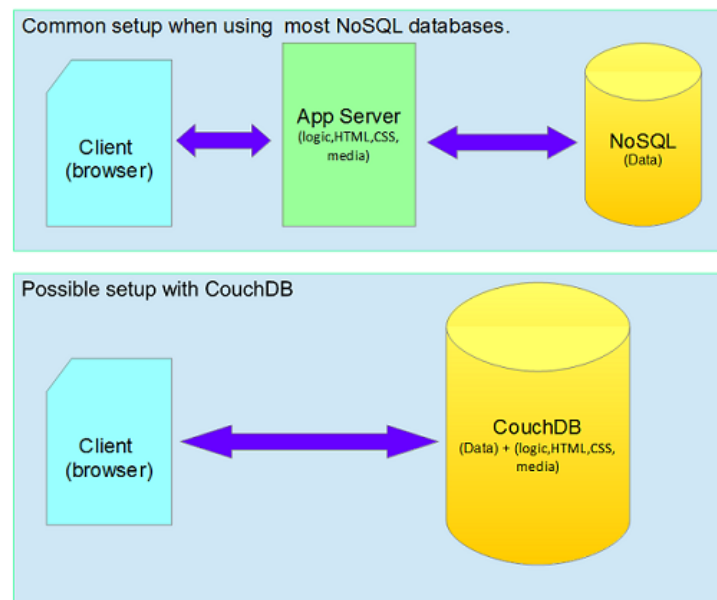


Figure 3. Different deployment configurations.

Json documents

CouchDB documents are represented in JSON format.

JSON is a document which can contain:

1. Object – represented by unordered set of key-value pairs.
2. Array – represented by ordered set of values.
3. Numbers,
4. Literals true, false, null.
5. String literal.

There are 3 preserved attributes:

1. `_id` – unique (for the server) value of a document. Mandatory.
2. `_rev` – unique (for the document) value identifying a certain revision of the document). Mandatory.
3. `deleted` – defines if the document has been deleted. Optional.

Http protocol

CouchDB uses HTTP protocol to perform communication between the database server and business applications. HTTP allow all 4 CRUD (Create, Read, Update, Delete) operations by using different HTTP requests:

Table 1. HTTP-SQL-CRUD correspondence.

Operation	SQL	HTTP
Create	INSERT	PUT/POST
Read	SELECT	GET
Update	UPDATE	POST/PUT/PATCH
Delete	DELETE	DELETE

Curl tool can be used as common HTTP interface helping to write request set scripts.

Table 2. Database list request.

Request	Response
<code>curl -X GET http://127.0.0.1:5984/_all_dbs</code>	<code>["social_network"]</code>

This request shows the list of all presenting database at the CouchDB server with the specified address.

A document can be uploaded by the following request:

Table 3. Document upload request.

Request	Response
<code>curl -X PUT http://localhost:5984/social/_design/find -d @find.txt</code>	<code>{"ok": true, "id": "_design/find", "rev": "1-6d536e5c381b91dc613342d3c567cff5"}</code>

The server returns success flag, document id and document version in JSON format. The same request can be applied in order to update a document. User should only specify the content of the new document. The document version will be automatically assigned by the server.

To access a document we can write a GET request:

Table 4. Document download request.

Request	Response
<code>curl -X GET http://localhost:5984/database/social/_design/find</code>	Document content in JSON

However since CouchDB is append-only database, the most correct way to delete a document is to upload a new version of the document with attribute “_deleted”

Listing 2. Document deleting revision.

```
{
  "_id": "mydoc",
  "some_data": 42,
  "_deleted": true
}
```

View, Map/Reduce

Accessing to the data is mostly done using views. View is composed of map and reduce function. Map function is written using JavaScript and has a single document as an input argument. Mapping function uses emit function to submit a key-value pair to the set of result pairs. Key and value can be represented by a single value or by a tuple of values. Reduce function is optional and is meant to summarize the result of map functions. Possible reducing is sum or count of the row values. It is also possible to rereduce the result in recursive manner.

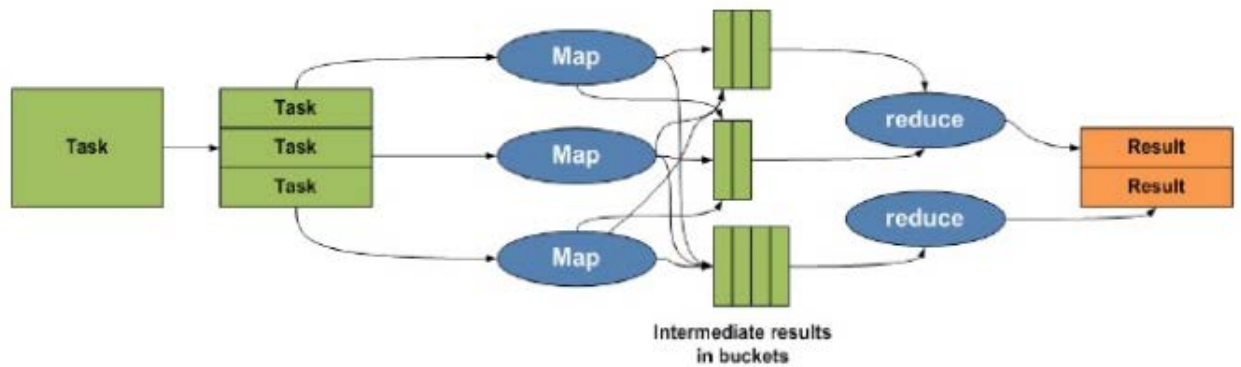


Figure 4. Map/Reduce sequence illustration.

Futon

Futon is a CouchDB full-functional web-interface tool for admins of the database. It is can be accessed by a web-browser.

Metadata	id	key	value
192f2278f1902ed67c4a3f7d...	192f2278f1902ed67c4a3f7d...	["aidar", "192f2278f1902ed6...	192f2278f1902ed67c4a3f7d...
192f2278f1902ed67c4a3f7d...	192f2278f1902ed67c4a3f7d...	["aidar", "192f2278f1902ed6...	192f2278f1902ed67c4a3f7d...
192f2278f1902ed67c4a3f7d...	192f2278f1902ed67c4a3f7d...	["najm", "192f2278f1902ed...	192f2278f1902ed67c4a3f7d...

Figure 5. Futon view result.

Social network developing

Database design

In order to perform database design in CouchDB we must accomplish following:

1. Elaborate model ER-diagram.
2. Design common documents based on the ER-diagram.
3. Design view documents based on the ER-diagram.

Model

We have arranged on the following ER-diagram model. The model implements following ideas:

1. Model has users.
2. Relationship defines the relationship between two users. It can be confirmed by one or both of the defined users.
3. There are groups in which users can participate.
4. Group_participation is an associative element between user and group.
5. Post is related with a user and may be related either to user's page or to some group. Contains text as content.
6. Comment is related to a user and to a post.

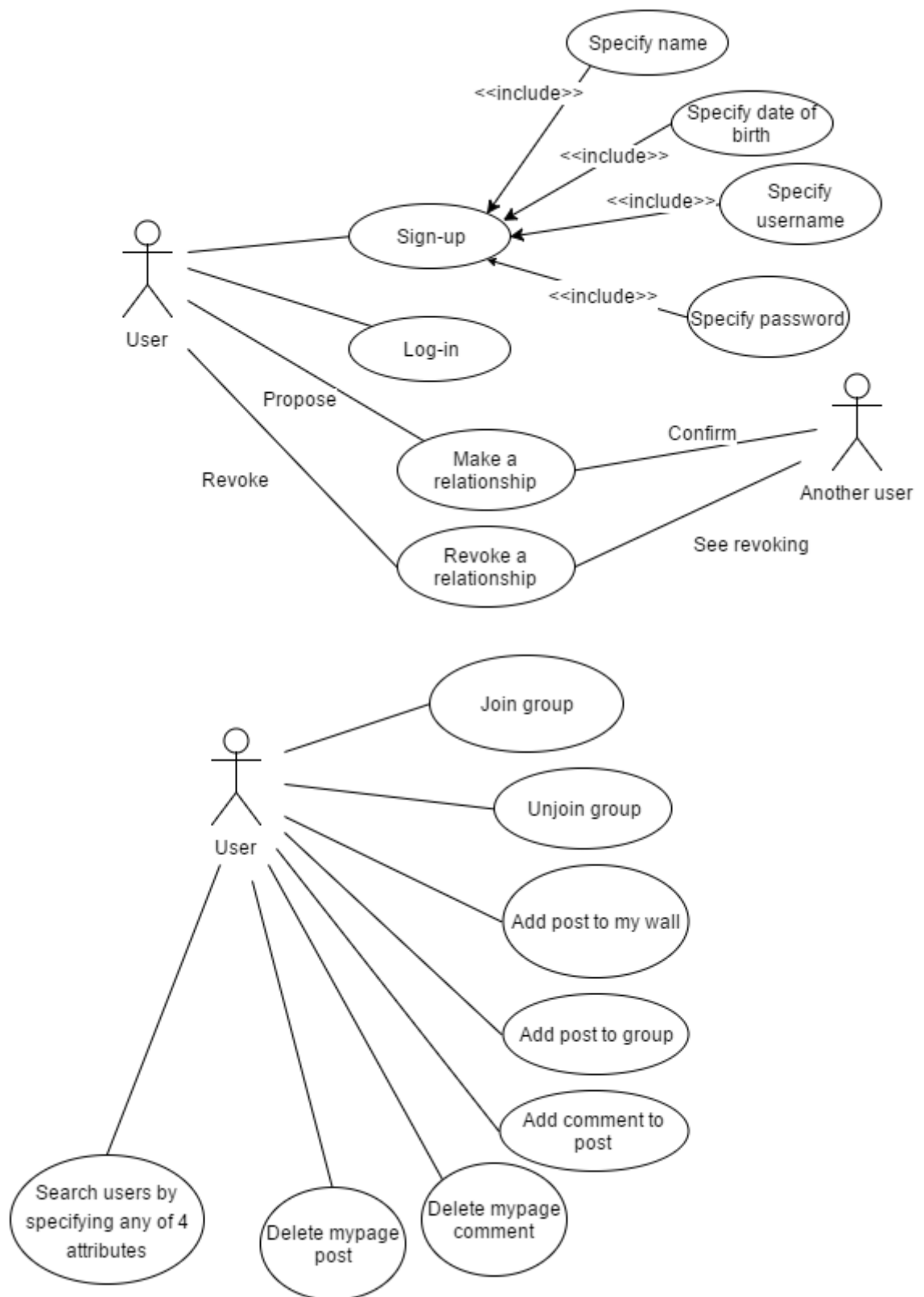


Figure 6. Use-case diagram.

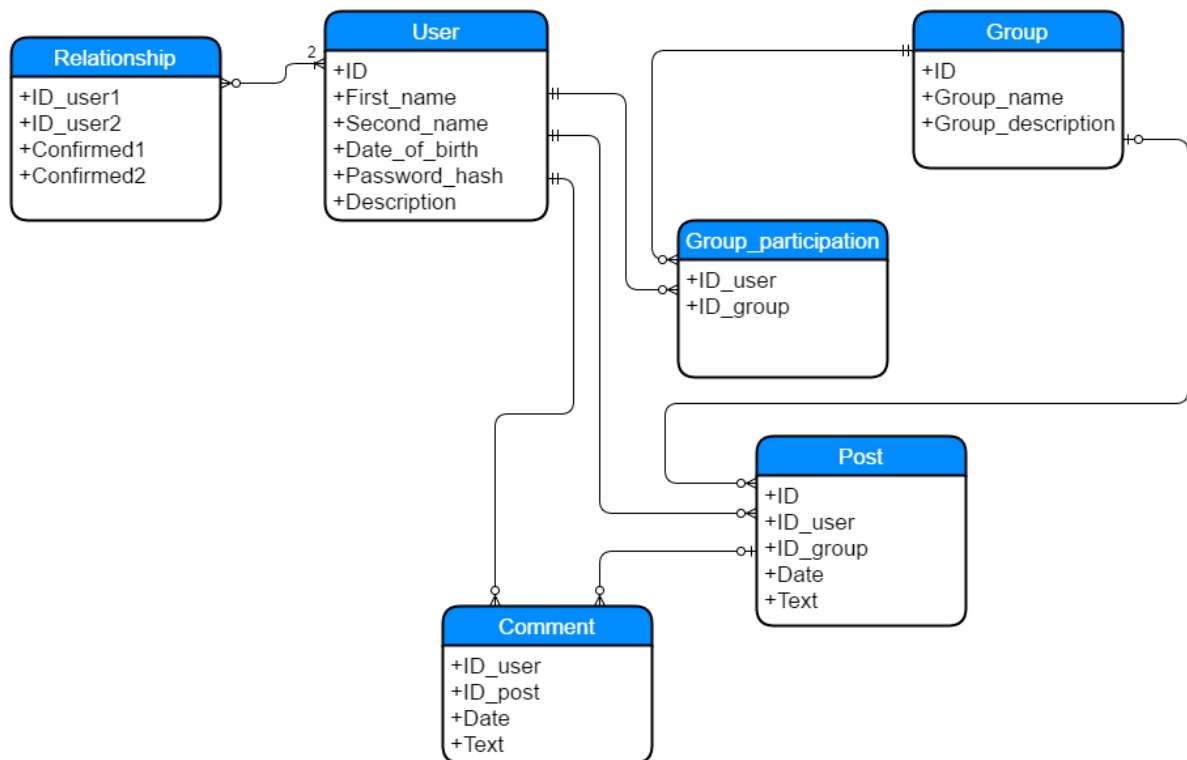


Figure 7. Social network ER-diagram.

In order to ensure the safety of the user password, we only store the hash value of the plaintext password, so the potential attacker won't be able to sign-in even knowing the hash due to a possible leak.

Common documents

The next thing we need to determine is how we decompose these entities over the documents. This is the point where CouchDB may show its power and flexibility. We can assign all objects of one type to a single document. We can assign every object to a separate document. We also can merge some objects of different types into one document. Here is our suggestion:

Table 5. Distribution of ER-entities over the document types.

	ER-entity	Document type
1.	User	User
2.	Relationship	Relationship
3.	Group	Group
4.	Group_participation	Group_participation
5.	Post	Post
6.	Comment	Post

In order to distinguish the documents of different types we can use an extra “type” attribute to each of the documents. As seen in the Table 5 we have merged comment objects into the post objects they are associated with. This could be done since in our social network all business logic working with the posts is also related to the comments. By this we attain some supplementary effectiveness because of removing need to build a search index for the comments.

View documents

We should design views to allow data access to any web-server requests. Views have been inferred based on what type of functionality is described in the use-case diagram. Following views have been developed:

Table 6. Find views.

Name	Map code	Reduce function
groups_by_id	<pre>function (doc) { if (doc.type == "group") emit(doc._id, doc) }</pre>	-
groups_by_username_id	<pre>function (doc) { if (doc.type == "participation") emit([doc.username, doc.group_id], doc.group_id) }</pre>	-
participation	<pre>function (doc) { if (doc.type == "participation") emit([doc.username, doc.group_id], doc._id);</pre>	-

	}	
post_by_group_date	function (doc) { if (doc.type=="post") if (doc.id_group!="-") emit([doc.id_group, doc.date], doc) }	-
post_by_username_date	function (doc) { if (doc.type=="post") if (doc.id_group=="-") emit([doc.username, doc.date], doc) }	-
relationship_confirmed_by_username	function (doc) { if (doc.type == "relationship") { if ((doc.confirmed1 == "yes") && (doc.confirmed2 == "yes")) { emit([doc.username1, doc.username2], doc._id) emit([doc.username2, doc.username1], doc._id) } } }	-
relationship_pending_by_username	function (doc) { if (doc.type == "relationship") { if ((doc.confirmed1 == "no") && (doc.confirmed2 == "yes")) emit([doc.username2, doc.username1], doc._id) if ((doc.confirmed1 == "yes") && (doc.confirmed2 == "no")) emit([doc.username1, doc.username2], doc._id) } }	-
relationship_proposed_by_username	function (doc) { if (doc.type == "relationship") { if ((doc.confirmed1 == "yes") && (doc.confirmed2 == "no"))	-

	<pre> emit([doc.username2, doc.username1], doc._id) if ((doc.confirmed1 == "no") && (doc.confirmed2 == "yes")) emit([doc.username1, doc.username2], doc._id) } </pre>	
user_by_bday _fname	<pre> function (doc) { if (doc.type == "user") emit([doc.date_of_birth, doc.first_name], doc) } </pre>	-
user_by_fname e_sname_bday	<pre> function (doc) { if (doc.type == "user") emit([doc.first_name, doc.second_name, doc.date_of_birth], doc) } </pre>	-
user_by_sname e_bday	<pre> function (doc) { if (doc.type == "user") emit([doc.second_name, doc.date_of_birth], doc) } </pre>	-
user_by_username	<pre> function (doc) { if (doc.type == "user") emit(doc.username, doc) } </pre>	-

And we also have introduced one statistics view which uses `_sum` reduce function.

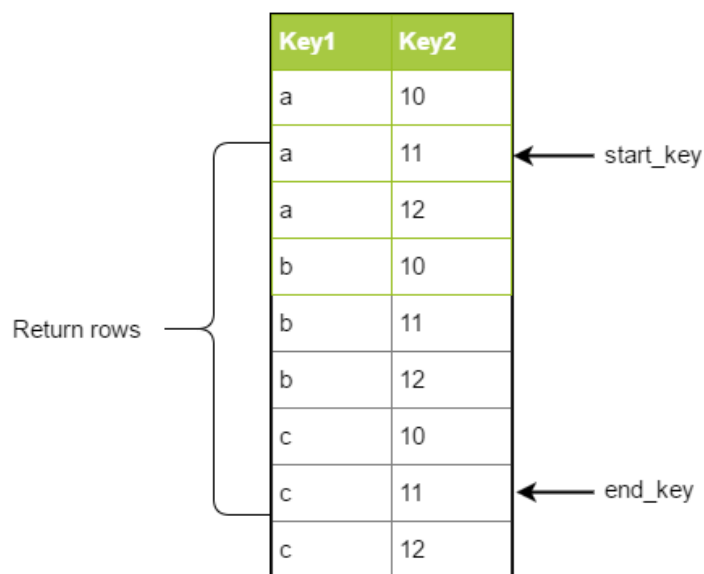
Table 7. Statistics views.

Name	Map code	Reduce function
users	<pre> function (doc) { if (doc.type == "user") emit(doc._id, 1) } </pre>	<code>_sum</code>

Reduce function sums the values of all the rows (which is 1 for every row) and therefore we obtain the number of the registered users in the system.

User search

Considering that the arranged functionality demands implementing of user search we had to implement several views (indexes) to allow user search on different input data. User entity has 4 attributes: username, first_name, second_name, birthday. We must remember that user may specify one attribute, some of them or all of them. One should remember the idea of how does CouchDB treats startkey and endkey parameters in case if tuple is used as key. If startkey == [a, 11] and endkey == [c, 11] then following rows will be returned:



Key1	Key2
a	10
a	11
a	12
b	10
b	11
b	12
c	10
c	11
c	12

Figure 8. Ranged key request result.

By definition in tupled key case the rows are ordered by the first key and in equals sectors it is ordered by the second key. Hence the range request returns uninterrupted range of rows which are more or equal than start_key and less or equal than end_key. In tuple key case the request may miss some of the ending elements but it has to contain the first ones.

Therefore we have developed following 4 views which can handle any search request by any attributes:

1. user_by_bday_fname.
2. user_by_fname_sname_bday.
3. user_by_sname_bday.
4. user_by_username.

Table 8. Search view selection.

Username is specified	FirstName is specified	SecondName is specified	Birthday is specified	Index
No	No	No	No	[Show all users]
No	No	No	Yes	user_by_bday_fname
No	No	Yes	No	user_by_sname_bday
No	No	Yes	Yes	user_by_sname_bday
No	Yes	No	No	user_by_fname_sname_bday
No	Yes	No	Yes	user_by_bday_fname
No	Yes	Yes	No	user_by_fname_sname_bday
No	Yes	Yes	Yes	user_by_fname_sname_bday
Yes	No	No	No	user_by_username
Yes	No	No	Yes	user_by_username
Yes	No	Yes	No	user_by_username
Yes	No	Yes	Yes	user_by_username
Yes	Yes	No	No	user_by_username
Yes	Yes	No	Yes	user_by_username
Yes	Yes	Yes	No	user_by_username
Yes	Yes	Yes	Yes	user_by_username

As we see since username is unique whenever the searcher specifies the username we can find this person (if exists) and simply check the fulfilling of other attributes matching. In all rest case we should user other views by providing all or part of the attributes as the keys.

Database establishing

In order to automatize the establishing of the initial database it was necessary to develop a set-up script using Curl tool [6].

Listing 3. Database establishing script.

```
curl -X DELETE http://127.0.0.1:5984/social
curl -X PUT http://127.0.0.1:5984/social
```

```
curl -X PUT http://localhost:5984/social/_design/find -d @find.txt
curl -X PUT http://localhost:5984/social/_design/statistics -d
@statistics.txt

curl -X PUT
http://127.0.0.1:5984/social/cff457c34484830b569a999a27014134 -d
@user_1.txt
curl -X PUT
http://127.0.0.1:5984/social/cff457c34484830b569a999a27016314 -d
@user_2.txt
...
```

Server description

Python was chosen as the language to develop the web-server. The web-interface was designed using Python CGI library [5]. Web-server and CouchDB server are located on the same host in our case; however it is possible to easily redistribute the system on several hosts.

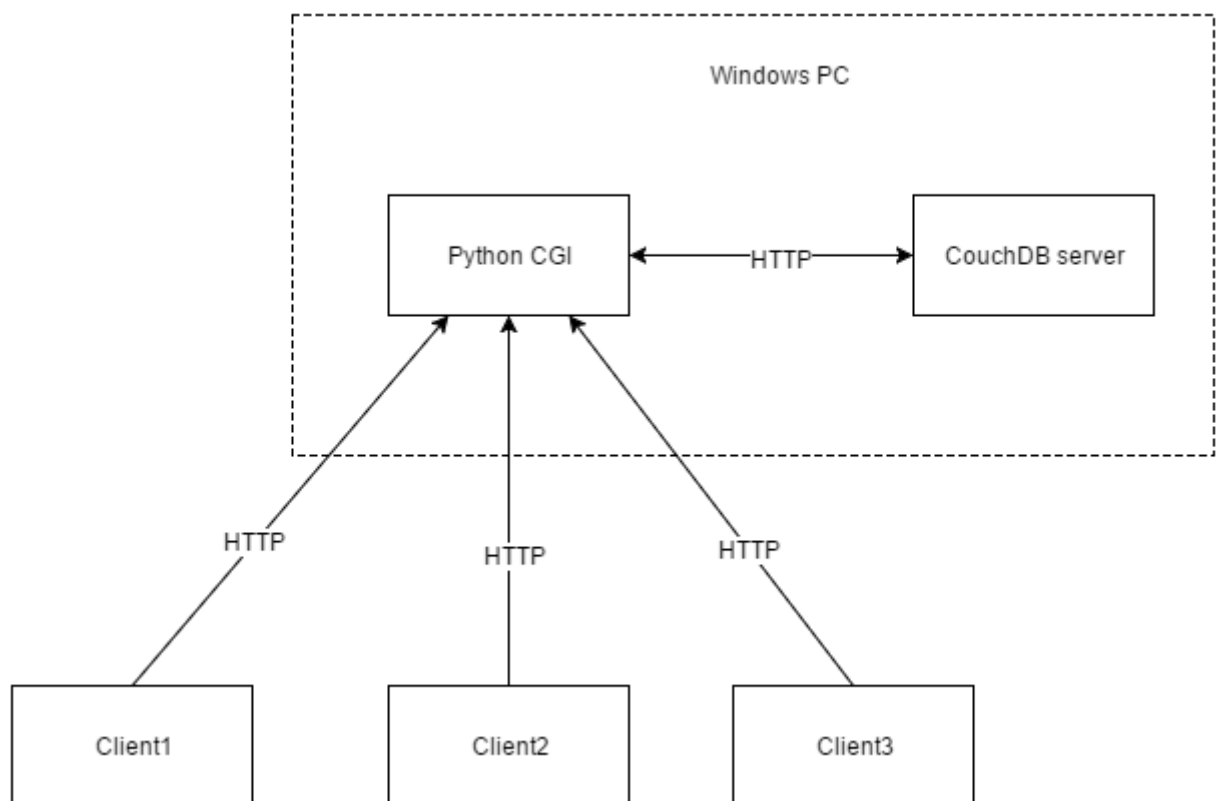


Figure 9. Server UML-deployment diagram.

Python DB access interface

During the development of the web-server backend one of the first problems we have encountered is creating a simple and convenient DB access interface. We have incapsulated HTTP GET and POST request into HttpApi class. Above the HttpApi we have Requests class every method of which any high-level user request which may require several low-level HTTP requests. Generally this class is a wrapper over the standard Python HTTP protocol library. Apparently GET request was used to obtain data from both common documents and view documents. POST request is used to create, update and delete documents.

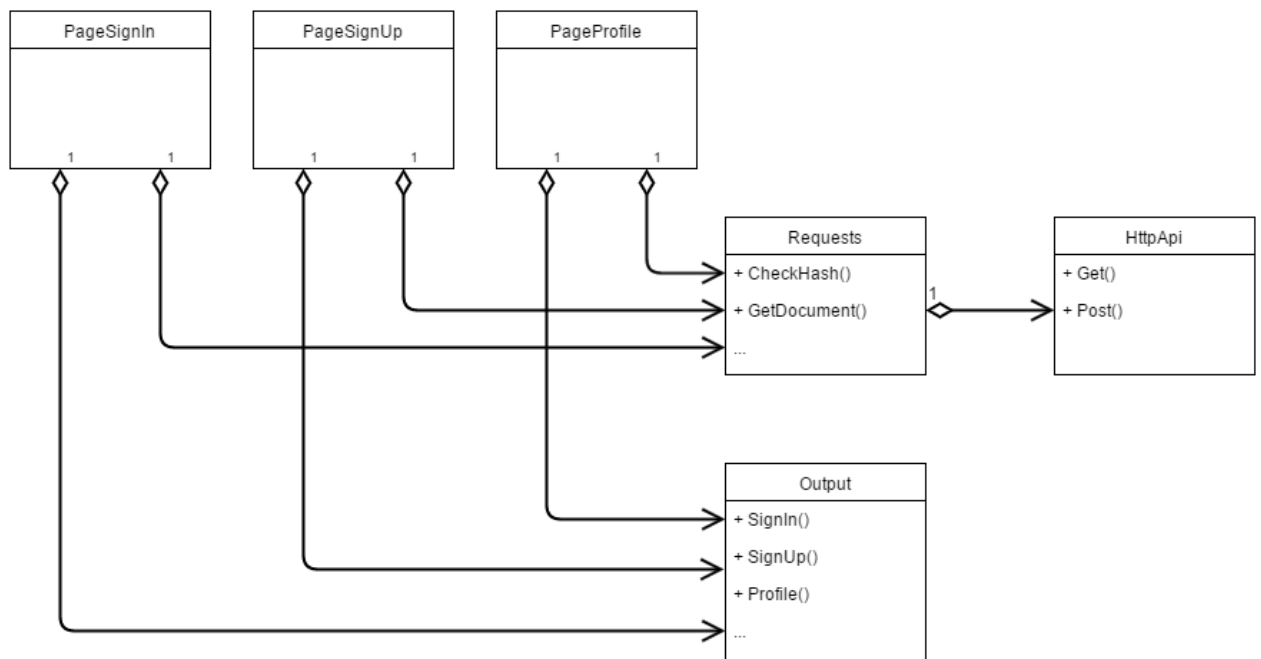


Figure 10. Web-server UML-class diagram.

Listing 4. POST and GET request wrappers.

```
@staticmethod
def GetDB(url):
    urlFull = Config.couchDbUrl + url
    response = urllib.request.urlopen(urlFull).read().decode("utf-8")
    return json.loads(response)

@staticmethod
def Get(url):
    response = urllib.request.urlopen(url).read().decode("utf-8")
    return json.loads(response)

@staticmethod
```

```
def PostDB(url, data):
    url_full = Config.couchDbUrl + url
    data = bytes(data.encode("utf-8"))
    req = urllib.request.Request(url_full, data, headers={"Content-Type": "application/json"})
    return urllib.request.urlopen(req)
```

Since CouchDB is an append-only database we may encounter a problem of overflowing the disk memory. In such case we must use compaction tool which is presented in CouchDB. Compaction removes the old revisions of the documents. Compaction can be called manually or it is possible to configure the compaction daemon into regular checking state. The second way is more preferable in our case since social network may consume many disk space in short time; so the admins won't have to conduct regular manual check of the disk space.

Listing 5. Compaction call request.

```
curl -H "Content-Type: application/json" -X POST
http://localhost:5984/social/_compact
```

Conclusion

The database analysis has been performed. The entity-relation model has been developed. Test database and server were successfully established in a local network (WLAN). Business logic of the server was implemented.

References

1. https://en.wikipedia.org/wiki/Web_2.0
2. <http://www.digizen.org/downloads/social-networking-overview.pdf>
3. <http://guide.couchdb.org/>
4. <https://wiki.apache.org/couchdb>
5. <https://docs.python.org/2/library/cgi.html>
6. https://www.tutorialspoint.com/couchdb/couchdb_curl_futon.htm
7. <https://automatthew.wordpress.com/2007/12/14/amazon-simplydb-and-couchdb-compared/>
8. <http://db-engines.com/en/system/CouchDB%3BRiak+KV>

Appendix



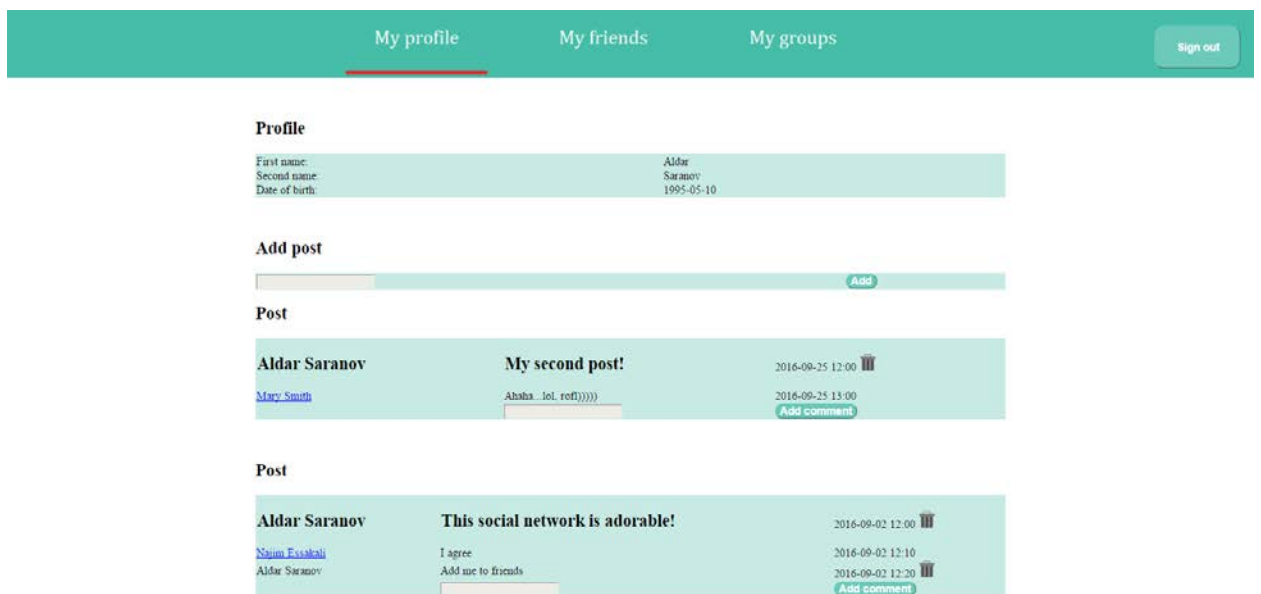
Sign up

Username

Password

Sign in

Figure 11. Sign-in page screenshot.



My profile My friends My groups Sign out

Profile

First name:	Aldar
Second name:	Saranov
Date of birth:	1995-05-10

Add post

Add

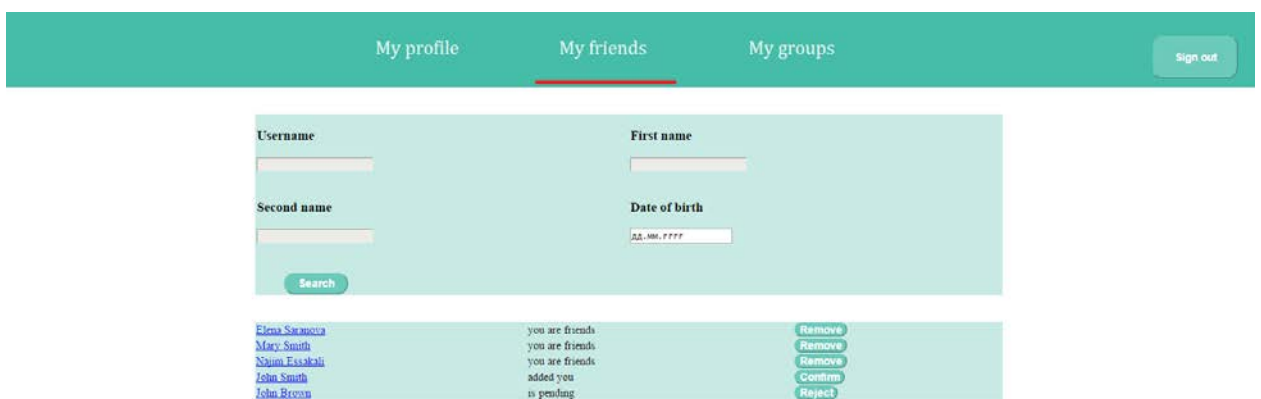
Post

Aldar Saranov	My second post!	2016-09-25 12:00	
Mary Smith	Ababa .lol. rofl))))))	2016-09-25 13:00	Add comment

Post

Aldar Saranov	This social network is adorable!	2016-09-02 12:00	
Najim Fayakali	I agree	2016-09-02 12:10	
Aldar Saranov	Add me to friends	2016-09-02 12:20	Add comment

Figure 12. Profile page screenshot.



My profile My friends My groups Sign out

Username First name

Second name Date of birth

Search

Elena Saranova	you are friends	Remove
Mary Smith	you are friends	Remove
Najim Fayakali	you are friends	Remove
John Smith	added you	Confirm
John Brown	is pending	Reject

Figure 13. My friends page screenshot.



Figure 14. Search page screenshot.



Figure 15. Group page screenshot.

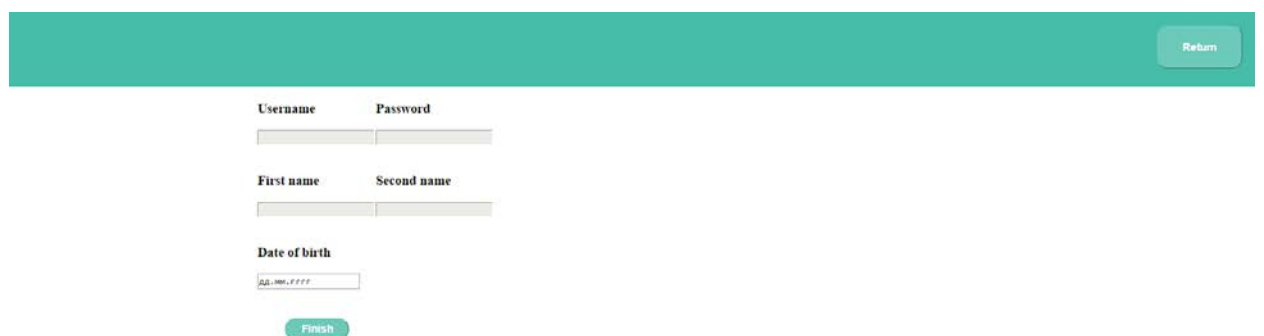


Figure 16. Sign-up page screenshot.