Group project (Databases)
Grinkevich, Demidova, Babichev BPI175

About program

This database is used in the web application for renting an apartment a long period. There are two types of users: tenants and landlords. The tenant views rental ads, filters and sorting them by price, location, area and number of rooms, sends a request to view the apartment, add rental ad to favorites list. The landlord creates, edits, deletes his rental ads, view requests from tenants and their phone numbers.

Requirements

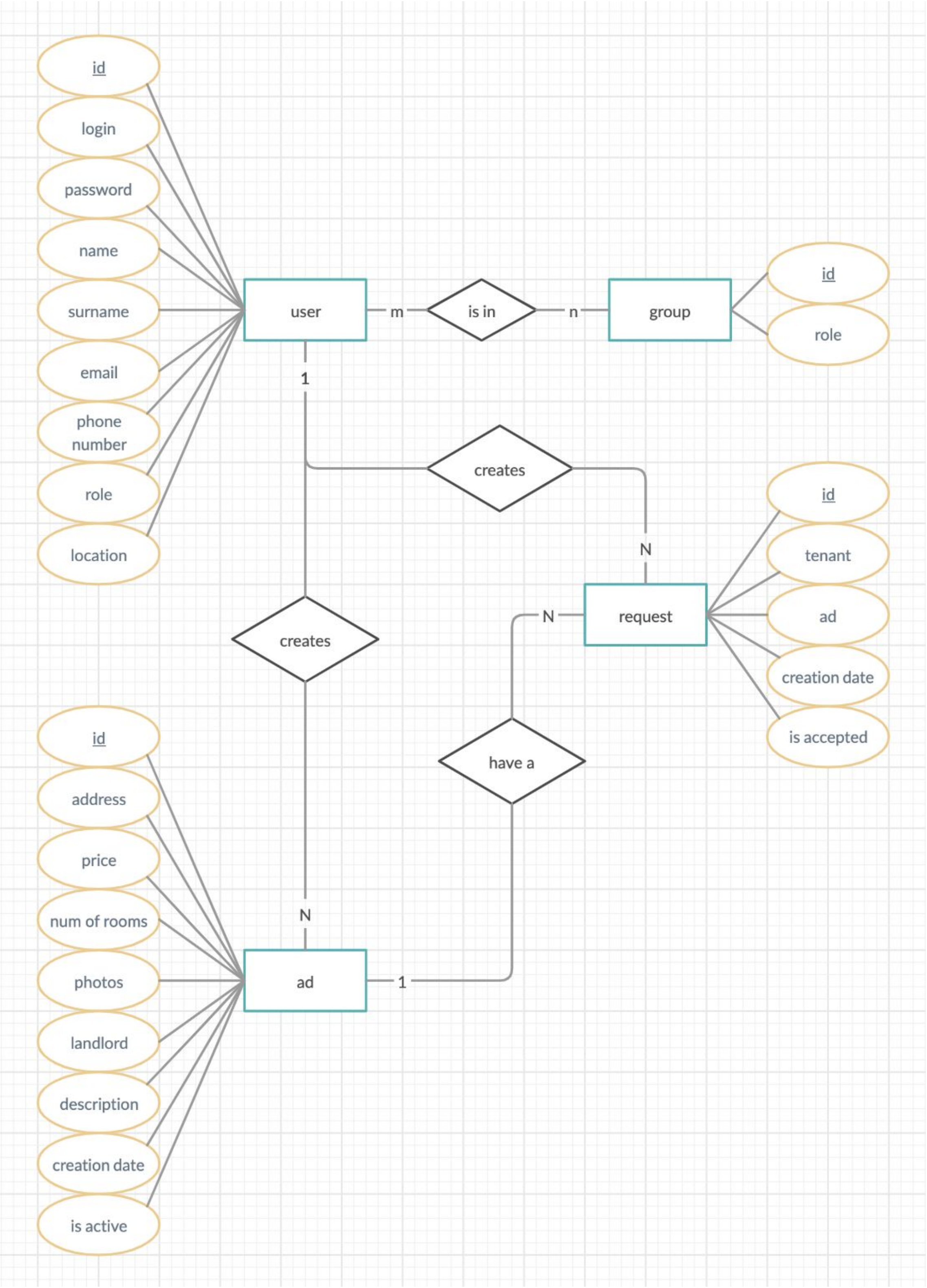
Functional:

1. Store information about users' accounts (tenants and landlords): phone number, login, password, email, name and surname, location, role.
2. Store information about rental ads: address, price, number of rooms, area, photos, description, landlord's name, creation date, association with landlord, activation flag.
3. Store information about requests: association with tenant, association with landlord, association with rental ad, date, accept/refuse flag.
4. CRUD for rental ads.
5. CRUD for requests.
6. Activate/deactivate ad.
7. Viewing, filtering and sorting ads by their parameters (for all users).
8. Accept/refuse request (for landlords).
9. Delete request (for tenant).
10. If user is a landlord, he still can make requests

Non-functional:

1. For each phone number can be only one tenant account and only one landlord account.

UML-diagram


id
login
password
name
surname
email
phone number
role
location
user
m
is in
n
group
id
role
1
creates
N
id
tenant
ad
creation date
is accepted
request
N
creates
have a
id
address
price
num of rooms
photos
landlord
description
creation date
is active
N
ad
1

Database creation

```
CREATE DATABASE LITRES;

CREATE TABLE User (
id int NOT NULL PRIMARY KEY,
name VARCHAR(255),
surname VARCHAR(255),
phone VARCHAR(255),
location VARCHAR(255),
login VARCHAR(255),
password VARCHAR(255),
email VARCHAR(255),
role_id int,
foreign key (role_id) references Group(id));

CREATE TABLE Group (
Id int NOT NULL PRIMARY KEY,
role VARCHAR(255));

CREATE TABLE Ad (
id int NOT NULL PRIMARY KEY,
address VARCHAR(255),
price int,
num_of_rooms int,
landlord_id int,
description VARCHAR(255),
creation_date DATETIME,
isActive boolean,
foreign key (landlord_id) references User(id));
```

```
CREATE TABLE Request (
id int NOT NULL PRIMARY KEY,
tenant_id int,
ad_id int,
creation_date DATETIME,
isAccepted boolean,
foreign key (landlord_id) references User(id),
foreign key (tenant_id) references User(id),
foreign key (ad_id) references Ad(id));

CREATE TABLE UserGroup (
user_id int NOT NULL PRIMARY KEY,
group_id int NOT NULL PRIMARY KEY);
```

Normalization

The database satisfies the first normal form because it executes next conditions:

1. No duplicate rows
2. All attributes are simple
3. All values are scalar

The database satisfies the second normal form because it executes conditions:

1. The table is in the first normal form
2. The table must have a primary key
3. All attributes should describe the primary key as a whole, not some part of the primary block

The database satisfies the third normal form because it executes conditions:

1. The table is in the second normal form
2. There should be no dependencies of some non-key attributes on others. All attributes depend only on the primary key.

Pros:

1. Normalization does our database more flexible.
2. But we can easily add something and it will not cause any conflicts.
3. Also there will be as few duplicates as possible, so this saves a lot of storage space.

Cons:

1. We have more tables & more relations.
2. Relations are more complex.
3. SQL requests sometimes are more complex than they could be.

SQL Queries

1) List of users who live in Russia:

   ```
   SELECT *
   FROM User
   WHERE location = “Russia”
   ```

2) Find all ads in Moscow with price under 30000 rubles.

   ```
   SELECT *
   FROM Ad
   WHERE price<30000 AND address LIKE ‘%Moscow%’
   ```

3) Find all ads in St.Petersburg with 3 rooms and price ascending

   ```
   SELECT *
   ```

```
FROM Ad
WHERE address LIKE ‘%St.Petersburg%’ AND num_of_rooms=3
ORDER BY price
```

4) Delete all requests to ads of landlord with id 12333

```
DELETE FROM Request
WHERE landlord_id=12333
```

5) Find the number of ads in Omsk

```
SELECT COUNT(*)
FROM Ad
WHERE address LIKE ‘%Omsk%’
```

6) Find the number of landlords in Ukraine

```
SELECT COUNT(*)
FROM User u, Group g
WHERE u.location=“Ukraine” AND g.role=”landlord”
```

7) Create new request from user with id 5555 to the ad with id 66666

```
INSERT INTO “Request” (tenant_id, ad_id, creation_date)
VALUES(5555, 66666, CURRENT_TIMESTAMP)
```

Denormalization

Pros:

1. Denormalization helps us to save some storage.

Cons:

1. If we need a complex request, in denormalized database it will be much more difficult to do.
2. It’s hard to join such tables.