Bilkent University

2017-2018 Spring Semester

CS 353 Database Systems Design Report

BilPlay - Group 22

Furkan Bacak

Enes Emre Erdem

Ikbal Kazar

Samil Iraz

https://github.com/ikbalkazar/CS353-Group-22

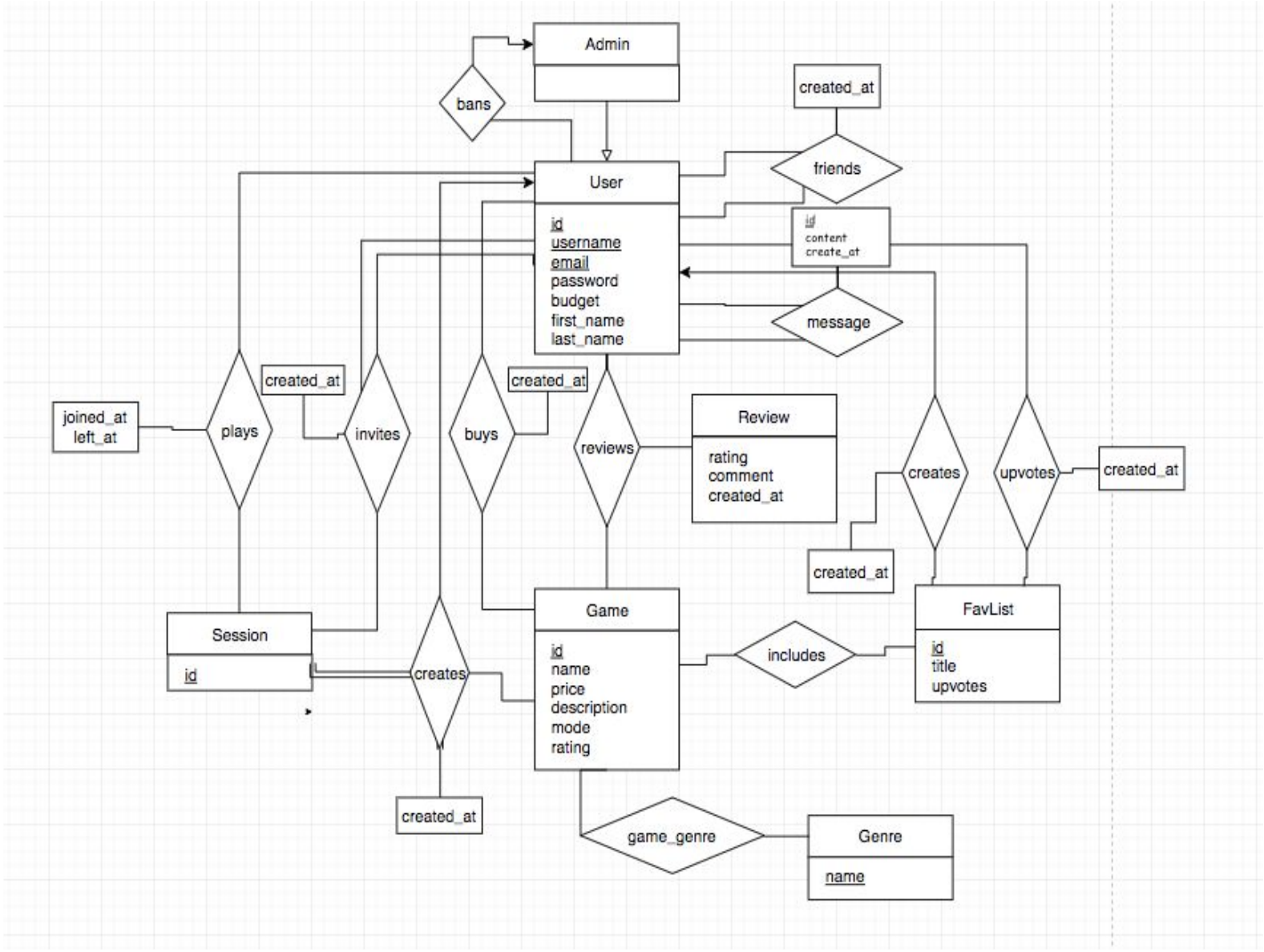
CONTENTS

1. Revised E/R Model

In accordance with the feedback we have received from our teaching assistant, we have made several changes and additions to our E/R model. The list of changes made and the resulting final E/R model can be found below.

- Added chat support for easier communication between friends to discuss recent games or check with friend's availability before sending an invite. This resulted in a "message" relationship between users in the E/R model.
- Modified "creates" relationship to make it a ternary relationship between User, Session and Game entities.

- Added a new feature that allows users to create favorite lists of games under a specific title. Favorite lists can be upvoted by users and the lists with the higher upvotes will be shown above in a feed. This resulted in “FavList” entity and “creates”, “upvotes”, “includes” relationships in the new E/R model.
- Added “joined_at” and “left_at” attributes to “plays” relationship which defines the period of time user spend in a game session.


Admin
bans
created_at
friends
User
id
username
email
password
budget
first_name
last_name
id
content
create_at
message
joined_at
left_at
plays
created_at
invites
buys
created_at
reviews
Review
rating
comment
created_at
creates
upvotes
created_at
created_at
Session
id
creates
Game
id
name
price
description
mode
rating
includes
FavList
id
title
upvotes
created_at
game_genre
Genre
name

2. Relation Schemas

1. **user**

Relational Model:

user(id, username, email, password, budget, first_name, last_name, is_admin, banned_by)

FK: banned_by ref. user(id)

Functional Dependencies:

id -> username, email, password, budget, first_name, last_name, is_admin, banned_by

username -> id, email, password, budget, first_name, last_name, is_admin, banned_by

email -> id, username, password, budget, first_name, last_name, is_admin, banned_by

Candidate Keys:

(id), (username), (email)

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `user` (
    `id` INT NOT NULL AUTO_INCREMENT,
    `username` VARCHAR(32) NOT NULL,
    `email` VARCHAR(128) NOT NULL,
    `password` VARCHAR(32) NOT NULL,
    `budget` DOUBLE NOT NULL DEFAULT 0,
    `first_name` VARCHAR(32),
    `last_name` VARCHAR(32),
    `is_admin` BOOL NOT NULL DEFAULT FALSE,
```

```
    `banned_by` INT,
    PRIMARY KEY (`id`),
    FOREIGN KEY (`banned_by`) REFERENCES `user`(id),
    UNIQUE (`username`),
    UNIQUE (`email`),
) Engine=InnoDB;
```

2. **friend**

   Relational Model:

   friend(user_id, friend_id, created_at)

   FK: user_id ref. user(id)

   FK: friend_id ref. user(id)

   Functional Dependencies:

   None

   Candidate Keys:

   (user_id, friend_id)

   Normal Form:

   BCNF

   Table Definition:

```
CREATE TABLE IF NOT EXISTS `friend` (
    `user_id` INT NOT NULL,
    `friend_id` INT NOT NULL,
    `created_at` DATETIME,
    PRIMARY KEY (`user_id`, `friend_id`),
```

```
    FOREIGN KEY (`user_id`) REFERENCES `user`(id),
    FOREIGN KEY (`friend_id`) REFERENCES `user`(id)
) Engine=InnoDB;
```

3. **invite**

   Relational Model:

   invite(id, user_id, friend_id, session_id, created_at)

   FK: user_id ref. user(id)

   FK: friend_id ref. user(id)

   FK: session_id ref. session(id)

   Functional Dependencies:

   id -> user_id, friend_id, session_id, created_at

   Candidate Keys:

   (id)

   Normal Form:

   BCNF

   Table Definition:

```
CREATE TABLE IF NOT EXISTS `invite` (
    `id` INT NOT NULL AUTO_INCREMENT,
    `user_id` INT NOT NULL,
    `friend_id` INT NOT NULL,
```

```
    `session_id` INT,
    `created_at` DATETIME,
    PRIMARY KEY (`id`),
    FOREIGN KEY (`user_id`) REFERENCES `user`(id),
    FOREIGN KEY (`friend_id`) REFERENCES `user`(id),
    FOREIGN KEY (`session_id`) REFERENCES `session`(id)
) Engine=InnoDB;
```

4. **user_session**

   Relational Model:

   user_session(user_id, session_id, joined_at, left_at)

   FK: user_id ref. user(id)

   FK: session_id ref. session(id)

   Functional Dependencies:

   (user_id, session_id) -> joined_at, left_at

   Candidate Keys:

   (user_id, session_id)

   Normal Form:

   BCNF

   Table Definition:

```
CREATE TABLE IF NOT EXISTS `user_session` (
    `user_id` INT NOT NULL,
```

```
    `session_id` INT NOT NULL,
    `joined_at` DATETIME,
    `left_at` DATETIME,
    PRIMARY KEY (`user_id`, `session_id`),
    FOREIGN KEY (`user_id`) REFERENCES `user`(id),
    FOREIGN KEY (`session_id`) REFERENCES `session`(id)
) Engine=InnoDB;
```

5. **game**

   Relational Model:

   game(id, name, price, mode, description, rating)

   Functional Dependencies:

   id -> name, price, mode, description, rating

   name -> id, price, mode, description, rating

   Candidate Keys:

   (id)

   Normal Form:

   BCNF

   Table Definition:

```
CREATE TABLE IF NOT EXISTS `game` (
    `id` INT NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(32) NOT NULL,
```

```
    `price` DOUBLE NOT NULL,
    `mode` INT NOT NULL,
    `description` VARCHAR(256),
    `rating` DOUBLE,
    PRIMARY KEY (`id`),
    Unique (`name`)
) Engine=InnoDB;
```

6. **session**

   Relational Model:

   session(id, creation_time, creator_id, game_id)

   FK: creator_id ref. user(id)

   FK: game_id ref. game(id)

   Functional Dependencies:

   id -> creation_time, creator_id, game_id

   Candidate Keys:

   (id)

   Normal Form:

   BCNF

   Table Definition:

```
CREATE TABLE IF NOT EXISTS `session` (
    `id` INT NOT NULL AUTO_INCREMENT,
```

```
    `creation_time` DATETIME,
    `creator_id` INT NOT NULL,
    `game_id` INT NOT NULL,
    PRIMARY KEY (`id`),
    FOREIGN KEY (`creator_id`) REFERENCES `user`(id),
    FOREIGN KEY (`game_id`) REFERENCES `game`(id)
) Engine=InnoDB;
```

7. **genre**

   Relational Model:

   genre(<u>name</u>)

   Functional Dependencies:

   None

   Candidate Keys:

   (name)

   Normal Form:

   BCNF

   Table Definition:

```
CREATE TABLE IF NOT EXISTS `genre` (
    `name` VARCHAR(32) NOT NULL,
    PRIMARY KEY (`name`)
) Engine=InnoDB;
```

8. **game_genre**

Relational Model:

game_genre(game_id, genre_name)

FK: game_id ref. game(id)

FK: genre_name ref. genre(id)

Functional Dependencies:

None

Candidate Keys:

(game_id, genre_name)

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `game_genre` (
    `game_id` INT NOT NULL,
    `genre_name` VARCHAR(32) NOT NULL,
    PRIMARY KEY (`game_id`, `genre_name`),
    FOREIGN KEY (`game_id`) REFERENCES `game`(id),
    FOREIGN KEY (`genre_name`) REFERENCES `genre`(id)
) Engine=InnoDB;
```

9. **purchase**

Relational Model:

purchase(user_id, game_id)

FK: user_id ref. user(id)

FK: game_id ref. game(id)

Functional Dependencies:

None

Candidate Keys:

(user_id, game_id)

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `purchase` (
    `user_id` INT NOT NULL,
    `game_id` INT NOT NULL,
    PRIMARY KEY (`user_id`, `game_id`),
    FOREIGN KEY (`user_id`) REFERENCES `user`(id),
    FOREIGN KEY (`game_id`) REFERENCES `game`(id)
) Engine=InnoDB;
```

10. **review**

Relational Model:

review(user_id, game_id, rating, comment, created_at)

FK: user_id ref. user(id)

FK: game_id ref. game(id)

Functional Dependencies:

user_id -> game_id, rating, comment, created_at

game_id -> user_id, rating, comment, created_at

Candidate Keys:

(user_id, game_id)

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `review` (
    `user_id` INT NOT NULL,
    `game_id` INT NOT NULL,
    `rating` INT NOT NULL,
    `comment` VARCHAR(32),
    `created_at` DATETIME,
    PRIMARY KEY (`user_id`, `game_id`),
    FOREIGN KEY (`user_id`) REFERENCES `user`(id),
    FOREIGN KEY (`game_id`) REFERENCES `game`(id)
) Engine=InnoDB;
```

11. **message**

Relational Model:

message(id, user_id, friend_id, content, created_at)

FK: user_id ref. user(id)

FK: friend_id ref. user(id)

Functional Dependencies:

id -> user_id, friend_id, content, created_at

Candidate Keys:

(id)

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `message` (
    `id` INT NOT NULL AUTO_INCREMENT,
    `user_id` INT NOT NULL,
    `friend_id` INT NOT NULL,
    `content` VARCHAR(256),
    `created_at` DATETIME,
    PRIMARY KEY (`id`),
    FOREIGN KEY (`user_id`) REFERENCES `user`(id),
    FOREIGN KEY (`friend_id`) REFERENCES `user`(id)
) Engine=InnoDB;
```

12. **favlist**

Relational Model:

favlist(id, creator_id, title, created_at)

FK: creator_id ref. user(id)

Functional Dependencies:

id -> creator_id, title, created_at

Candidate Keys:

(id)

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `favlist` (
    `id` INT NOT NULL AUTO_INCREMENT,
    `creator_id` INT NOT NULL,
    `title` VARCHAR(128) NOT NULL,
    `created_at` DATETIME,
    PRIMARY KEY (`id`),
    FOREIGN KEY (`creator_id`) REFERENCES `user`(id)
) Engine=InnoDB;
```

13. **favlist_upvote**

Relational Model:

favlist_upvote(user_id, favlist_id, created_at)

FK: user_id ref. user(id)

FK: favlist_id ref. favlist(id)

Functional Dependencies:

(user_id, favlist_id) -> created_at

Candidate Keys:

(user_id, favlist_id)

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `favlist_upvote` (
    `user_id` INT NOT NULL,
    `favlist_id` INT NOT NULL,
    `created_at` DATETIME,
    PRIMARY KEY (`user_id`, `favlist_id`),
    FOREIGN KEY (`user_id`) REFERENCES `user`(id),
    FOREIGN KEY (`favlist_id`) REFERENCES `favlist`(id)
) Engine=InnoDB;
```

14. **favlist_game**

Relational Model:

favlist_game(favlist_id, game_id)

FK: favlist_id ref. favlist(id)

FK: game_id ref. game(id)

Functional Dependencies:

None

Candidate Keys:

(favlist_id, game_id)

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `favlist_game` (
    `favlist_id` INT NOT NULL,
    `game_id` INT NOT NULL,
    PRIMARY KEY (`favlist_id`, `game_id`),
    FOREIGN KEY (`favlist_id`) REFERENCES `favlist`(id),
    FOREIGN KEY (`game_id`) REFERENCES `game`(id)
) Engine=InnoDB;
```

3. Functional Components

3.1 Use Cases

3.1.1 User

- **Change Password:** Users will be able to change their passwords.
- **Send Message:** Users can send the direct messages to others.
- **Read Message:** Users can read the direct messages that sent by others.
- **Add Friends:** Users can add other users as their friends.
- **Play a Game:** Users can play game.
- **Create a Game Session:** Users will be able to create a game session to play with their friends together.
- **Join a Game Session:** Users can join a game session that created by another user.
- **Invite Other Users to Game Session:** The creator users of game sessions can invite other users to play together.
- **Buy Games:** Users can buy new games with their currents.
- **Review the Game:** Users can write their reviews about games that owned by them.

3.1.2 Admin

- **Change Password:** Admins will be able to change their passwords.
- **Send Message:** Admins can send the direct messages to others.

- **Read Message:** Admins can read the direct messages that sent by others.
- **Add Friends:** Admins can add other users as their friends.
- **Play a Game:** Admins can play game.
- **Create a Game Session:** Admins will be able to create a game session to play with their friends together.
- **Join a Game Session:** Admins can join a game session that created by another user.
- **Invite Other Users to Game Session:** The creator admins of game sessions can invite other users to play together.
- **Buy Games:** Admins can buy new games with their currents.
- **Review the Game:** Admins can write their reviews about games that owned by them.
- **Ban User:** Admins can ban users depending on some reasons such as cheating, harassment etc.

3.2 Algorithms

There are two types of algorithms that we will use in our project. Our project is mainly about games and users. To make our project clear and safe, we will need 3 general algorithms. First algorithm is about the relationship between users such as adding each other as their friend or sending messages. When user addes another user as his/her friend, there will be changes in the tables. And in this adding process, blocks must be checked because users may block each other to protect them. Second algorithm is about the games and users. There will be some actions like reviewing a game or buying games. When one of these actions happened, some tables and informations in the database needs to be updated. The third algorithm will

be about general adding and deleting actions. For example to these actions, new games and new users can be given. In these adding processes, variables such as names must be checked because same names cant be used by multi games or people. These new things require updates on the tables.

3.3 Data Structures

We use Numeric Types, String Types and Date Types in tables of our database.

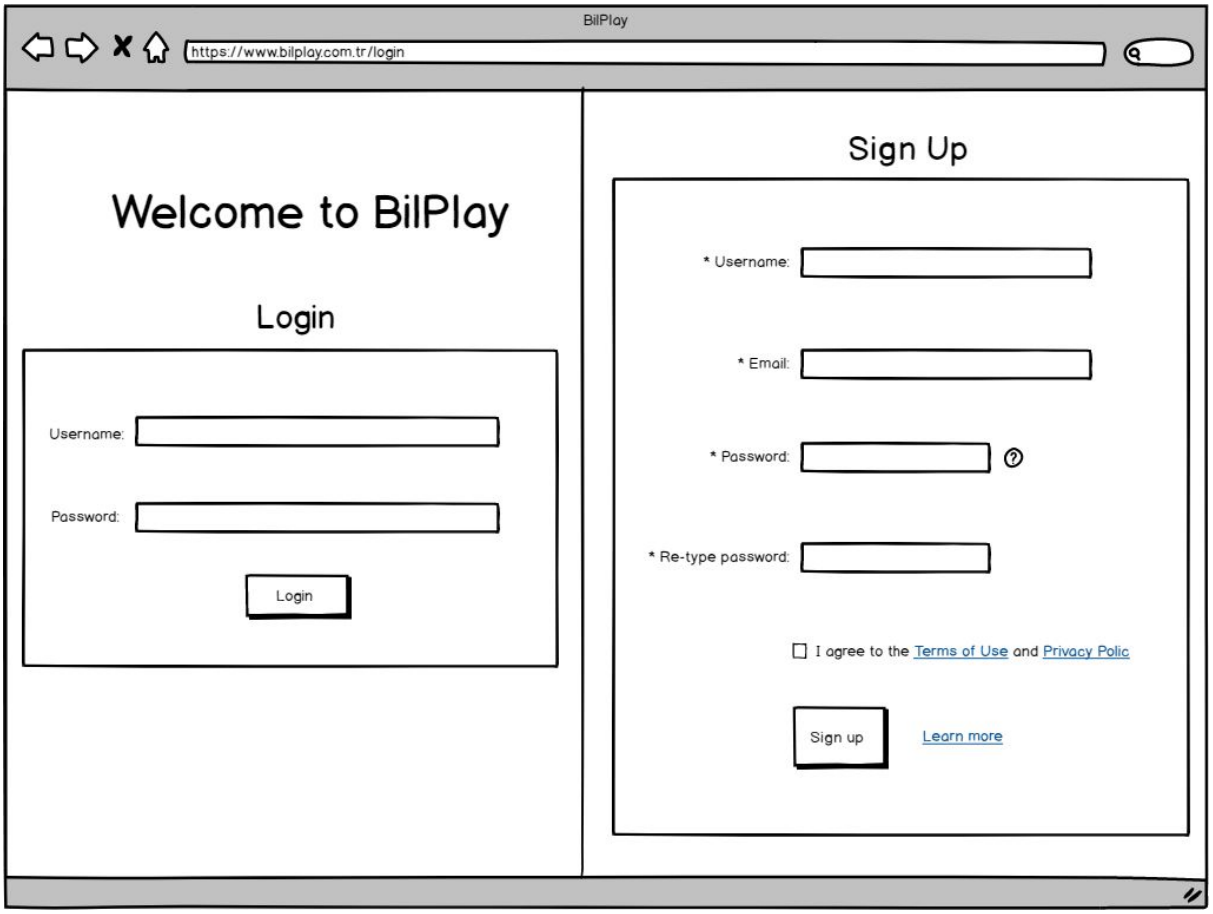
Numeric types are used to keep numbers such as budgets of users or values of games. We use INT, DOUBLE as numeric types in our database.

String types are used to keep information such as names, descriptions and messages in our database. We use VARCHAR, TINYTEXT as string types in our database.

Date and time types are used to keep date and times values of some informations such as time of messages. For this type, we use TIMESTAMP in our database.

4. User Interface Design and SQL Statements

4.1 Login and Register Screen



Inputs: @username, @password

Process: Users will login to their accounts by entering their username and passwords.

SQL Statements:

SELECT id

FROM User

WHERE username= @username AND password = @password

Inputs: @username, @password, @repassword, @email

Process: Users will register for an account by entering their email address, preferred password and non-taken username.

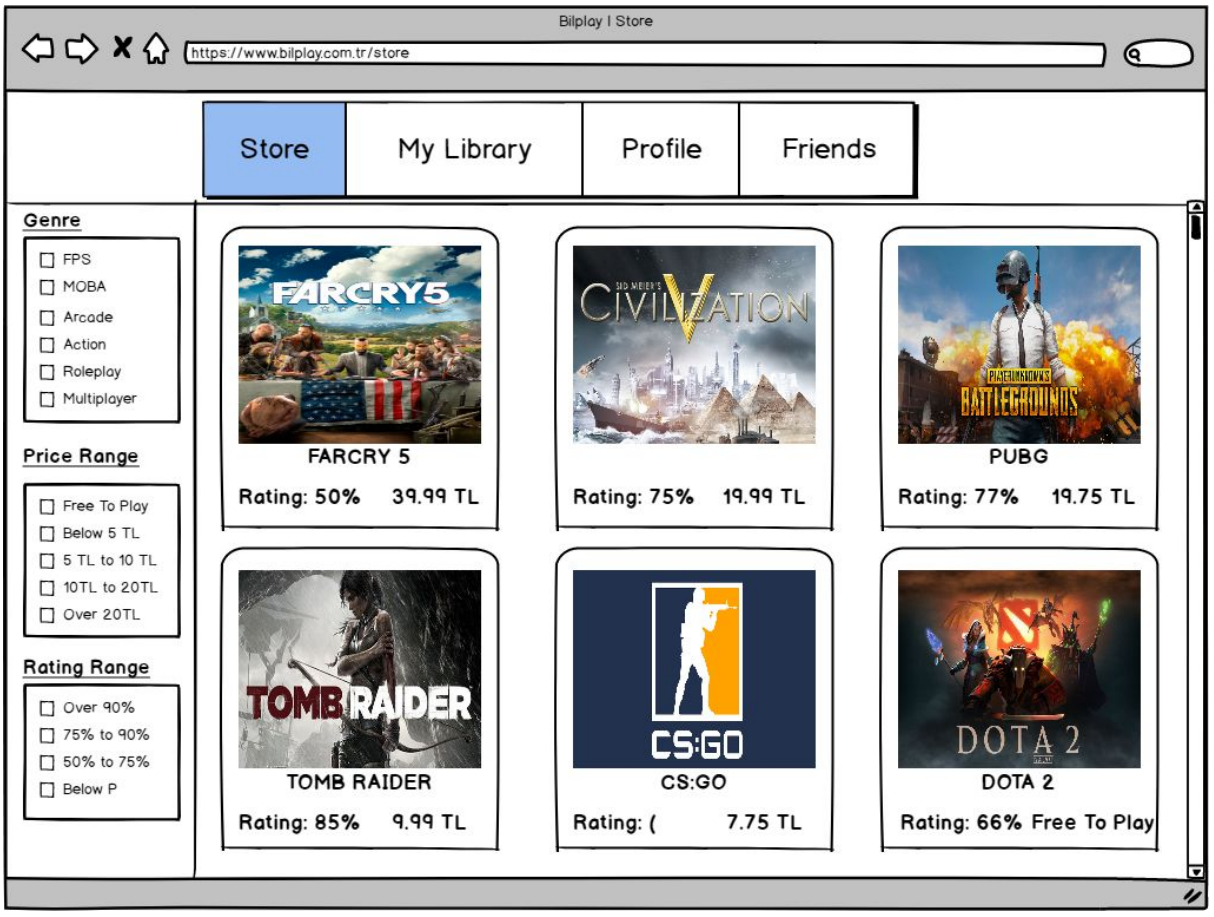
SQL Statements:

INSERT INTO User (id, username, email, password, budget, is_admin)

VALUES (@id, @username, @email, @password, '0.00', '0')

WHERE @password = @repassword

4.2 Store Screen



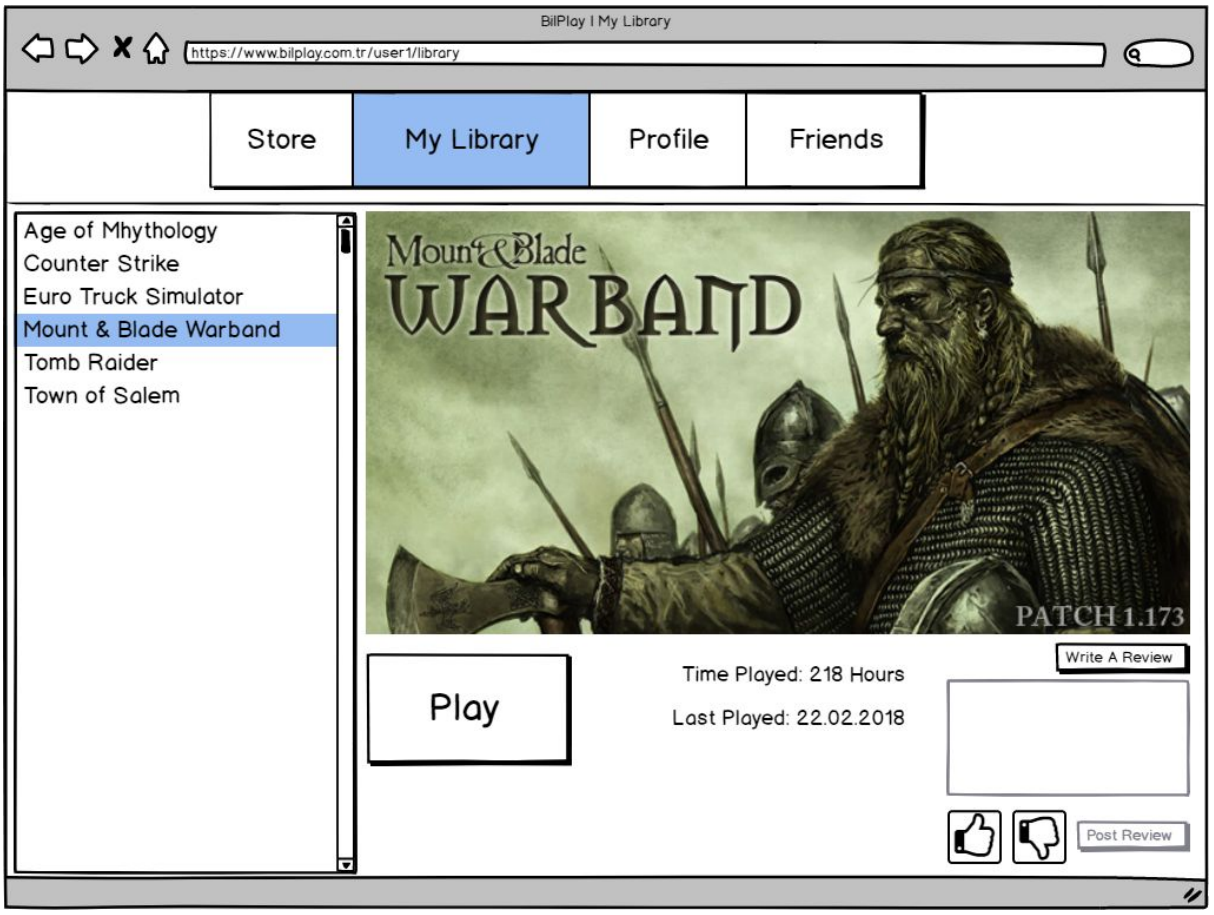
Inputs: table @price_range, table @rating_range, table @wished_genres

Process: Users can choose many options from each group: Genre, Price Range, Rating Range. With the options chosen by the user, database provides the appropriate games and displays them with details (name, price, rating)

SQL Statements:

```
WITH wanted_games_genre AS ( SELECT DISTINCT game_id FROM games_genre, wished_genres WHERE genre = genre_name ),
    wanted_games_price AS ( SELECT DISTINCT game_id FROM games, price_range WHERE price = price_value ),
    wanted_games_rating AS ( SELECT DISTINCT game_id FROM games, rating_range  WHERE rating = rating_value )
SELECT name, price, rating
FROM game, (
    SELECT wanted_games_rating_id
    FROM wanted_games_rating, (
        SELECT wanted_games_genre_id
        FROM wanted_games_genre, wanted_games_price
        WHERE wanted_games_genre_id = wanted_games_price_id )
    WHERE wanted_games_genre_id = wanted_games_rating_id )
WHERE id = wanted_games_rating_id
ORDER BY rating DESC
```

4.3 My Library Screen



Inputs: @user_id

Process: Users can see the games they own in their library. Library provides games in ascending order by their names. Users can choose any game to play. This page also shows users playtime and last played date of the selected game.

SQL Statements:

```
SELECT name
FROM game , (       SELECT game_id
                    FROM purchase
                    WHERE user_id = @user_id )
WHERE id = game_id
ORDER BY name ASC
```

Inputs: @user_id, @comment, @rating, @game_name, @date_time

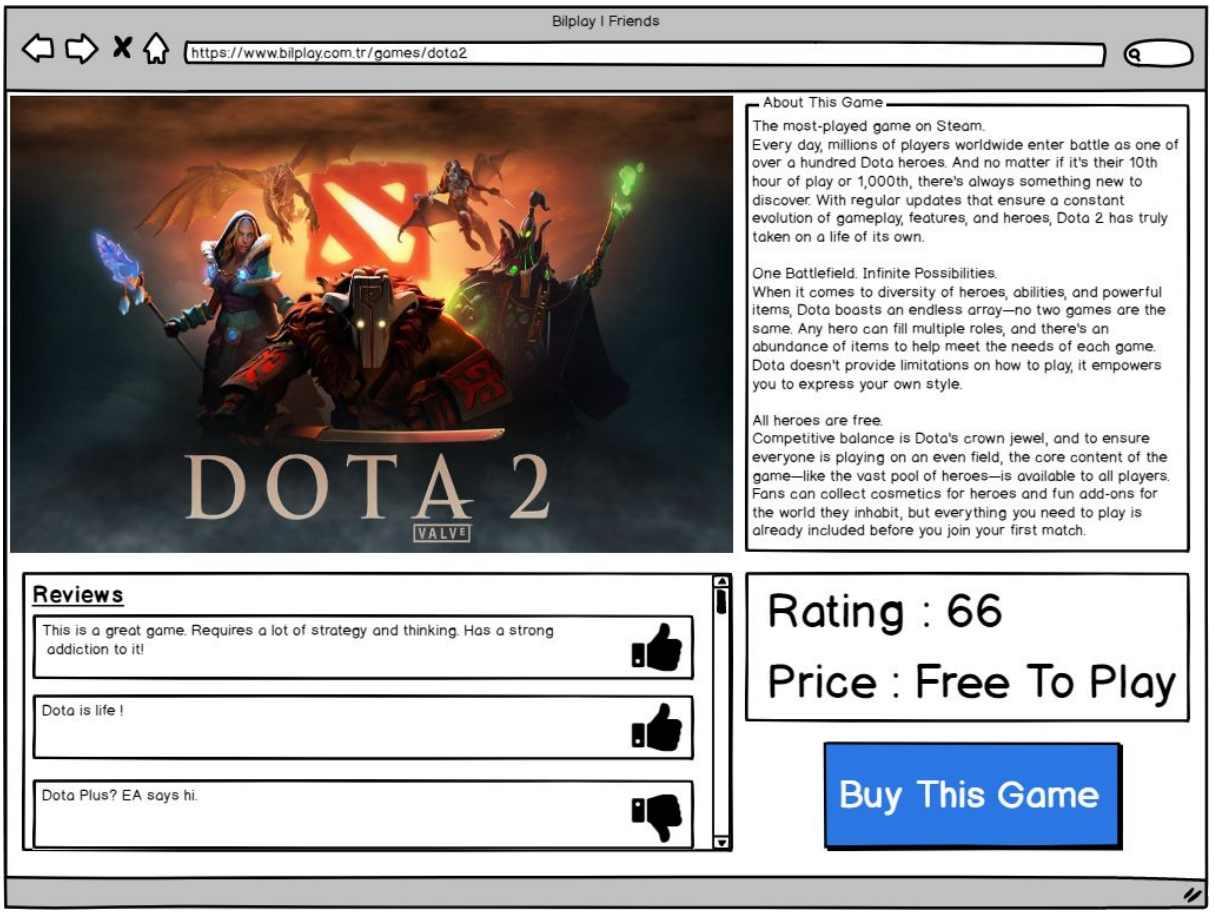
Process: This part is also about “My Library” page. When users click on “Write Review” button, comment section becomes active and users are able to write their own thoughts about the selected game and post whether they like it or not.

SQL Statements:

INSERT INTO review

VALUES (@user_id, (SELECT id FROM game WHERE @game_name = name), @rating, @comment, @date_time)

4.4 Game Page



Inputs: @game_id

Process: Upon clicking on a game in Store Screen, the will be directed to this Game Page where further details regarding the game will be displayed.

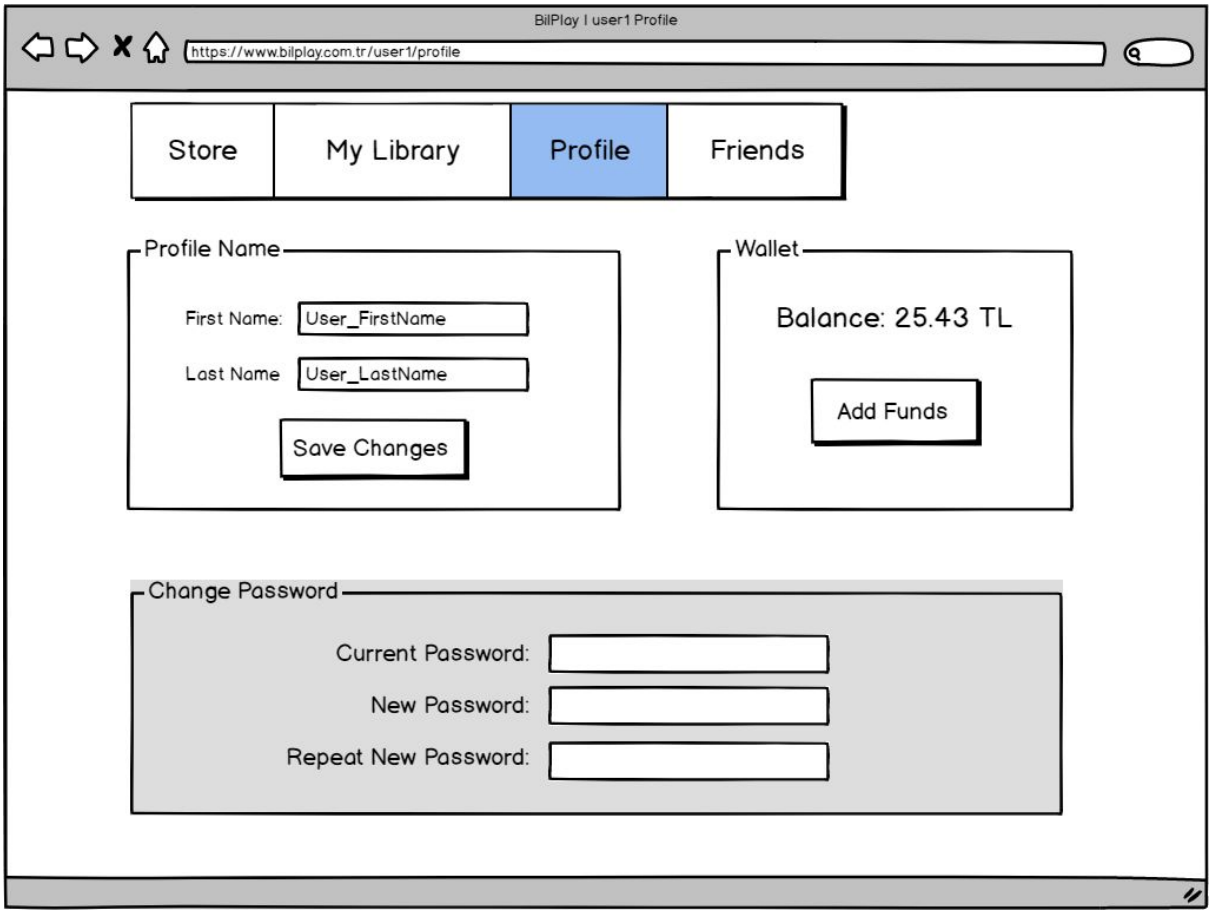
SQL Statements:

SELECT *

FROM game

WHERE id = @game_id

4.5 User Profile



Inputs: @user_id, @first_name, @last_name

Process: The user can update his first name and last name.

SQL Statements:

UPDATE User

SET first_name = @first_name, last_name = @last_name

WHERE user_id = @user_id

Inputs: @user_id, @current_password, @new_password, @re_new_password

Process: The user can change his password upon request. New password can't be the same with the current password.

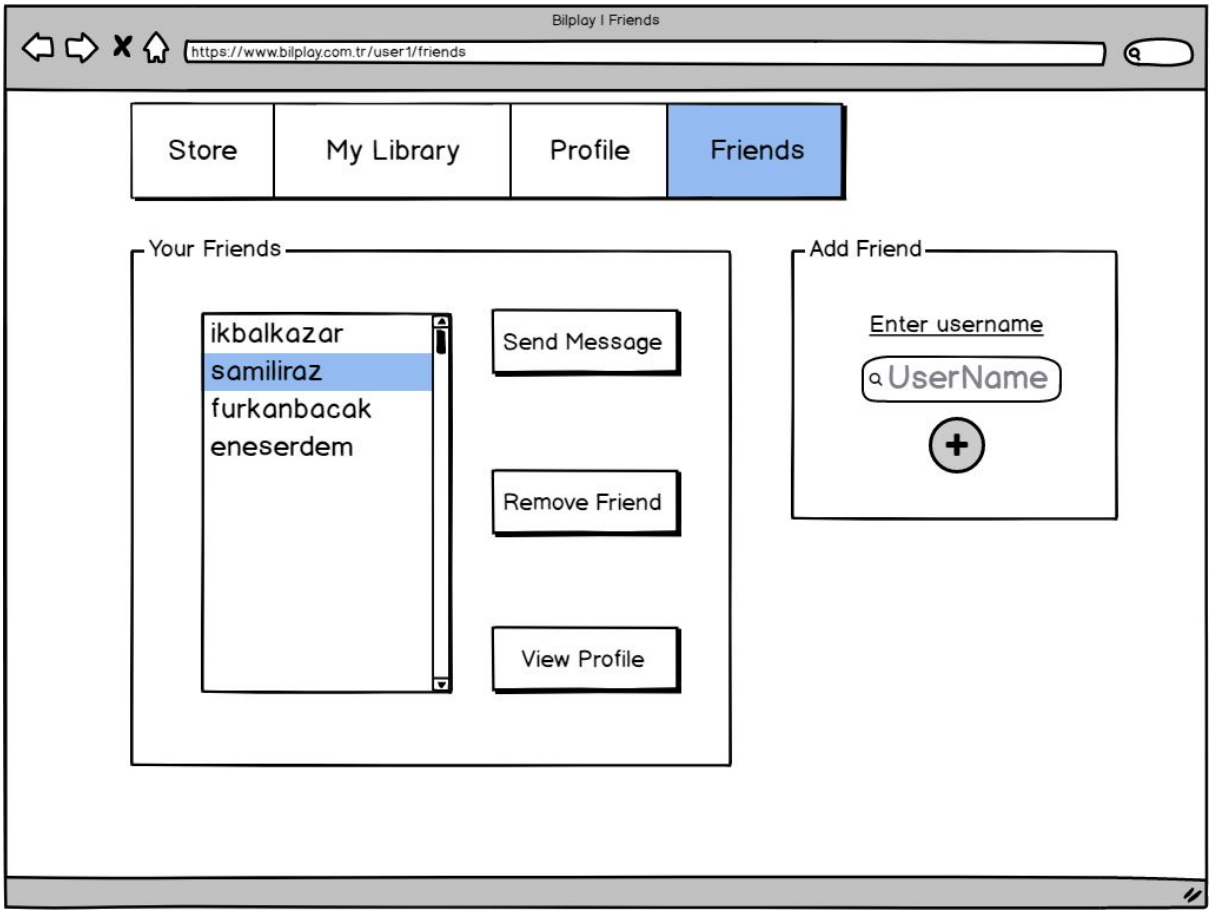
SQL Statements:

UPDATE User

SET password = @new_password

WHERE (user_id = @user_id AND @new_password = @re_new_password AND password = @current_password AND password != new_password)

4.6 Friends



Inputs: @user_id

Process: Retrieving the friend list of the user.

SQL Statements:

```
SELECT username
FROM user NATURAL JOIN (SELECT friend_id AS id
                        FROM friend
                        WHERE user_id = @user_id)
ORDER BY username ASC
```

Inputs: @user_id, @username, @date_time

Process: Adding a friend to friends list.

SQL Statements:

```
INSERT INTO friend
VALUES (@user_id, (SELECT id FROM User
                   WHERE username = @username), @date_time)
```

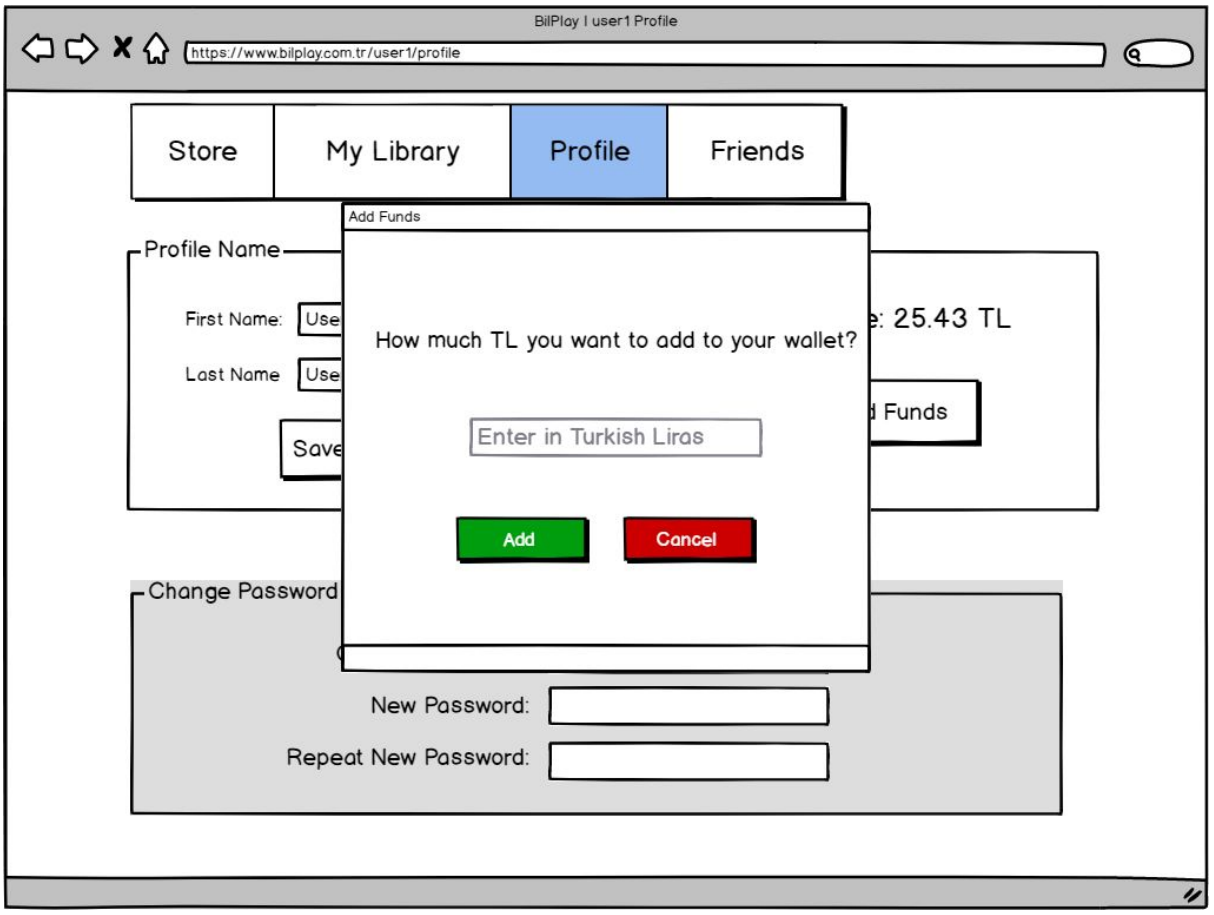
Inputs: @user_id, @friend_id

Process: Removing a friend from friends list.

SQL Statements:

```
DELETE FROM friend
WHERE user_id = @user_id, friend_id = @friend_id
```

4.7 Add Funds Pop-up



Inputs: @user_id, @money_Added

Process: The user can add money to wallet.

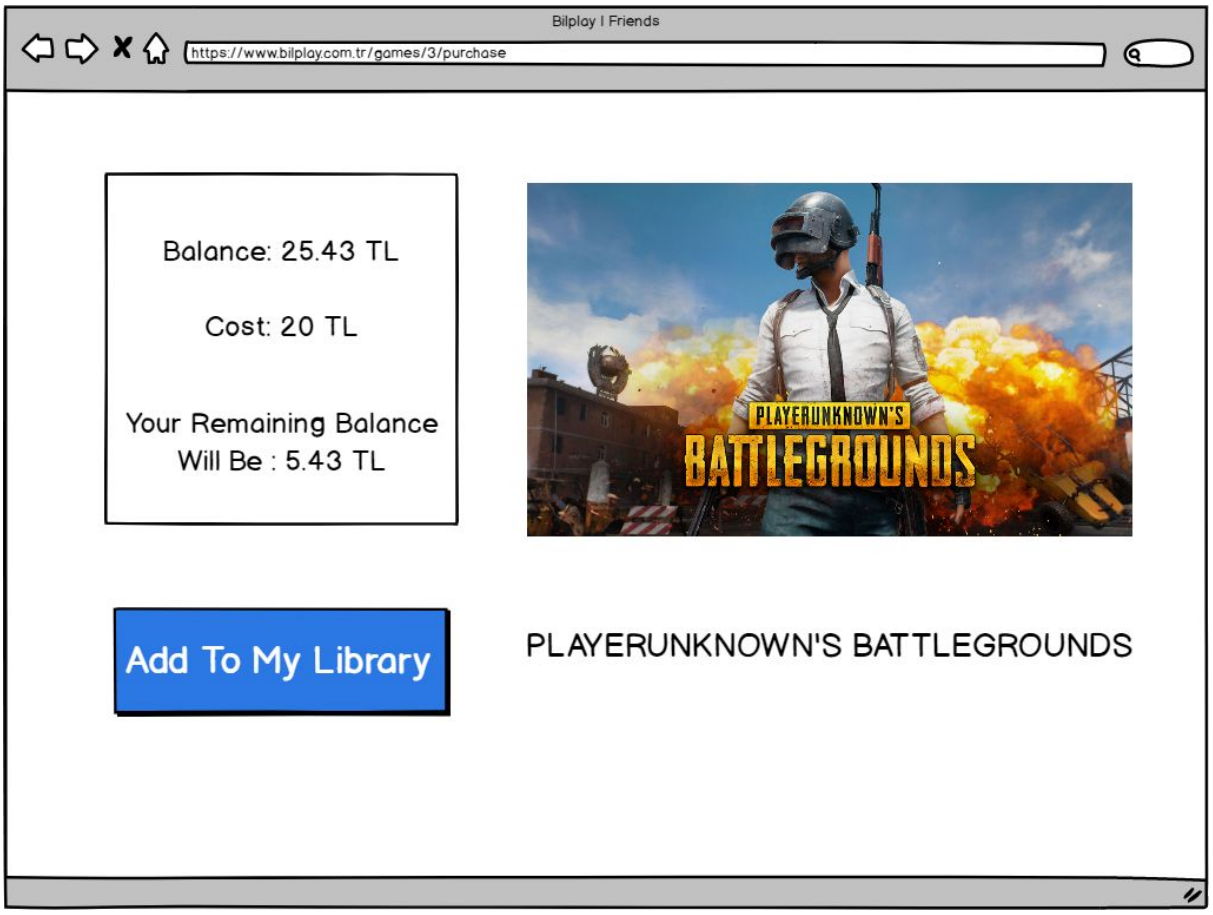
SQL Statements:

UPDATE User

SET budget = budget + @money_Added

WHERE id = @user_id

4.8 Purchase Game



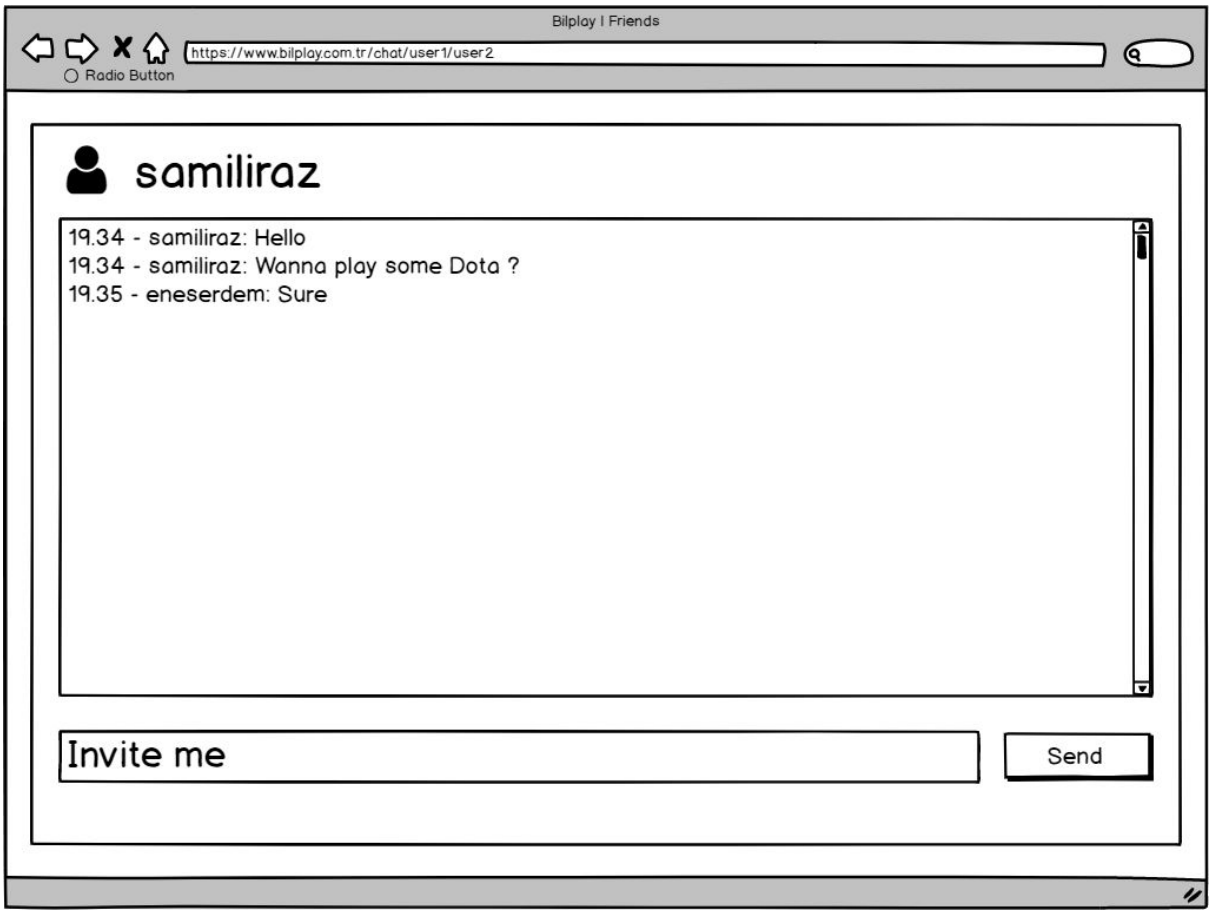
Inputs: @user_id ,@game_id

Process: When user clicks the button “Add To My Library” The game is added to the users game library and his the cost of the game will be dropped from user’s budget.

SQL Statements:

```
UPDATE User
SET budget = budget - (SELECT price
                       FROM Game
                       WHERE id = @game_id)
WHERE id = @user_id;
INSERT INTO purchase
VALUES (@user_id, @game_id)
```

4.9 Chat Screen



Inputs: @user_id**,** @friend_id

Process: User can see all messages with a friend.

SQL Statements:

SELECT * FROM message WHERE (user_id = @user_id AND friend_id = @friend_id) OR (user_id = @friend_id AND friend_id = @user_id);

Inputs: @user_id**,** @friend_id, @content

Process: User sends a message to a friend.

SQL Statements:

INSERT INTO message (user_id, friend_id, content) VALUES (@user_id, @friend_id, @content);

4.10 Favourite Games Lists

Inputs: @list_id

Process: The games that are included in a specific favourite games list is displayed in this page. Each list is belong to a specific user.

SQL Statements: SELECT name FROM FavList,Game

WHERE game_id = Game.id and FavList.id = @list_id;

Inputs: @user_id, @list_id

Process: User can upvote a favorite list.

SQL Statements: INSERT INTO favlist_upvote (user_id, favlist_id) VALUES (@user_id, @list_id);

5. Advanced Database Components

5.1 Views

5.1.1 Messages View

Users will be able to see messages that they sent or are sent to them. However, they can not see other messages that do not concern them.

CREATE VIEW view_message

SELECT * from message

WHERE user_id = @user_id OR friend_id = @friend_id

5.1.2 Game View By Name

Users will be able to information of games such as genres, description value etc.

CREATE VIEW view_game_name as

SELECT * from game NATURAL JOIN game_genre

WHERE name = @name AND game_id = id

5.1.3 Review View

Users can see and check the reviews of the game to decide whether buy or not.

CREATE VIEW view_review as

SELECT * from game NATURAL JOIN review

WHERE name = @name

5.1.4 View Game by Genre

Users may search games by genres such as FPS, Strategy.

CREATE VIEW view_game_genre as

SELECT * from game NATURAL JOIN game_genre

WHERE game_id = id AND @game_genre = game_genre

5.2 Stored Procedures

5.2.1 Get games list of a user

DROP PROCEDURE IF EXISTS getGamesListUser;

delimiter//

CREATE DEFINER = `root` @ `localhost` **PROCEDURE** ` getGamesListUser ` (param INT)

BEGIN

SET @s = `**SELECT** name **FROM** purchase NATURAL JOIN game **WHERE** id = game_id AND user_id = @user_id`;

SELECT @s;

PREPARE stmt FROM @s;

EXECUTE stmt;

END

delimiter;

5.2.2 Get friends list of a user

DROP PROCEDURE IF EXISTS getFriendsListUser;

delimiter//

CREATE DEFINER = `root` @ `localhost` **PROCEDURE** ` getFriendsListUser ` (param INT)

BEGIN

SET @s = `**SELECT** first_name, last_name **FROM** user, (**SELECT** friend_id **FROM** friend NATURAL JOIN user **WHERE** id = user_id AND id = @id`) **WHERE** id = friend_id ;

SELECT @s;

PREPARE stmt FROM @s;

EXECUTE stmt;

END

delimiter;

5.2.3 Get games of favorite list

DROP PROCEDURE IF EXISTS getGamesFavoriteList;

delimiter//

CREATE DEFINER = `root` @ `localhost` **PROCEDURE** ` getGamesFavoriteList ` (param INT)

BEGIN

SET @s = `**SELECT** name **FROM** game, favlist_game **WHERE** @favlist_id = favlist_id **AND** id = game_id`;

SELECT @s;

PREPARE stmt FROM @s;

EXECUTE stmt;

END

delimiter;

5.3 Reports

5.3.1 Total Number of Users

SELECT count (*)

FROM user

5.3.2 Total Number of Games

SELECT count(*)

FROM game

5.3.3 Total Number of Genres

SELECT count(*)

FROM genre

5.3.4 Total Number of Reviews of each Games

SELECT name AS game_name, count(*) AS number_reviews

FROM game NATURAL JOIN review

GROUP BY name

5.4 Triggers

- When a user is removed from the system, his/her messages must be deleted too.
- When a user is removed from the system, she/he must be deleted from the friend lists of other users.
- When new genre is added to genre list, all games must be updated according to this genre.
- When all players in the game session are leaved, this game session should be destroyed and removed its whole information from the tables.
- When users remove one of their friend, this removed friend's list must be updated.
- When user removed or updated his review and rate on game, game's informations also must be updated according to user action.

5.5 Constraints

- Users have to login to system to play games.
- While registering the system, username must be checked whether it is used or not.
- Users can play only the owned games by them.
- Users can write their reviews to only owned games by them.
- Users can create game sessions on only owned game by them.
- Users can invite other users who have game session's game to game session.
- Adding friend is double-sided. Both side must accept this relationship.
- Games at least have one genre type.
- Games at least have one mode type.
- Users can send messages just to their friends.
- Users can buy games depending on their budgets. They can not exceed their budgets.
- Users can rate their owned games not the other games.

6. Implementation Plan

We are going to use MySQL to implement our designed database with InnoDB engine. We will implement a REST API in Java and connect to database from our Java code with JDBC interface which will provide prepared sql statements and connection to MySQL. We will implement a static single page web application in javascript and html for the front-end which will be communicate with back-end through HTTP.