Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Базы данных

Курсовая работа
Разработка API для базы данных

Работу выполнил:
Калашников Р.А.
Группа:
3530901/70203
Преподаватель:
Мяснов А.В.

Санкт-Петербург
2020

Содержание

1. Постановка задачи

1.1. Цель

Создать RESTful API для базы данных, разработанной в рамках курса лабораторных работ, а также Android-приложение, которое будет его использовать.

1.2. Используемые средства

Сервер было решено делать на node.js, а для его тестирования использовать Postman. В качестве библиотеки, используемой для взаимодействия с сервером, был выбран Retrofit2.

1.3. Реализуемая функциональность

API должен предоставлять возможность авторизации пользователей, просмотра и покупки игр и планов по подписке, запись игровых сессий, а также просмотр информации о них. Android-приложение, общающееся с сервером, должно предоставлять пользователю интерфейс для авторизации, просмотра списков доступных и приобретенных игр и планов по подписке, а также симуляции игровых сессий и просмотра информации о них.

1.4. План работы

1. Определиться с возможностями API;
2. Создать и протестировать сервер API;
3. Создать и протестировать Android-приложение.

2. Подготовка

2.1. База данных

В данной работе используется база данных для стримингового сервиса игр, написанная в рамках курса лабораторных работ. Проект доступен по ссылке `http://gitlab.icc.spbstu.ru/kalashnikov.ra/videogame-streaming-service`.

Таблицы базы данных:

- client - содержит информацию о клиентах. Атрибуты:
  - nickname - имя пользователя, может быть не уникальным;
  - hash - хэш пароля;
  - email - почтовый адрес пользователя, должен быть уникальным.
- game - содержит информацию об играх
  - title - название игры;
  - price - цена игры.
- genre - содержит жанры игр
  - name - название жанра.

- subscription_plan - содержит информацию о планах по подписке
  - name - название плана;
  - price - цена плана за месяц.
- machine - содержит информацию о компьютерах
  - power_tier - мощность компьютера.
- owned_game - содержит информацию о приобретенных играх
  - client_id - id клиента, который приобрел игру;
  - game_id - id приобретенной игры;
  - purchase_date - дата приобретения.
- game_genre - содержит информацию о жанрах игр
  - game_id - id игры;
  - genre_id - id жанра.
- installed_game - содержит информацию об установленных на компьютерах играх
  - machine_id - id компьютера;
  - game_id - id игры.
- client_subscription_plan - содержит информацию о клиентских планах по подписке
  - client_id - id клиента;
  - subscription_plan_id - id плана подписки;
  - active_from - дата начала действия плана;
  - active_to - дата окончания действия плана.
- machine_usage - содержит информацию об использовании компьютеров
  - owned_game_id - id приобретенной игры;
  - machine_id - id компьютера;
  - in_use_from - время начала использования компьютера;
  - in_use_to - время окончания использования компьютера.
- available_machine_tier - содержит информацию о доступности компьютеров для планов по подписке
  - subscription_plan_id - id плана подписки;
  - machine_id - id компьютера;

Схема базы представлена на рисунке 2.1.

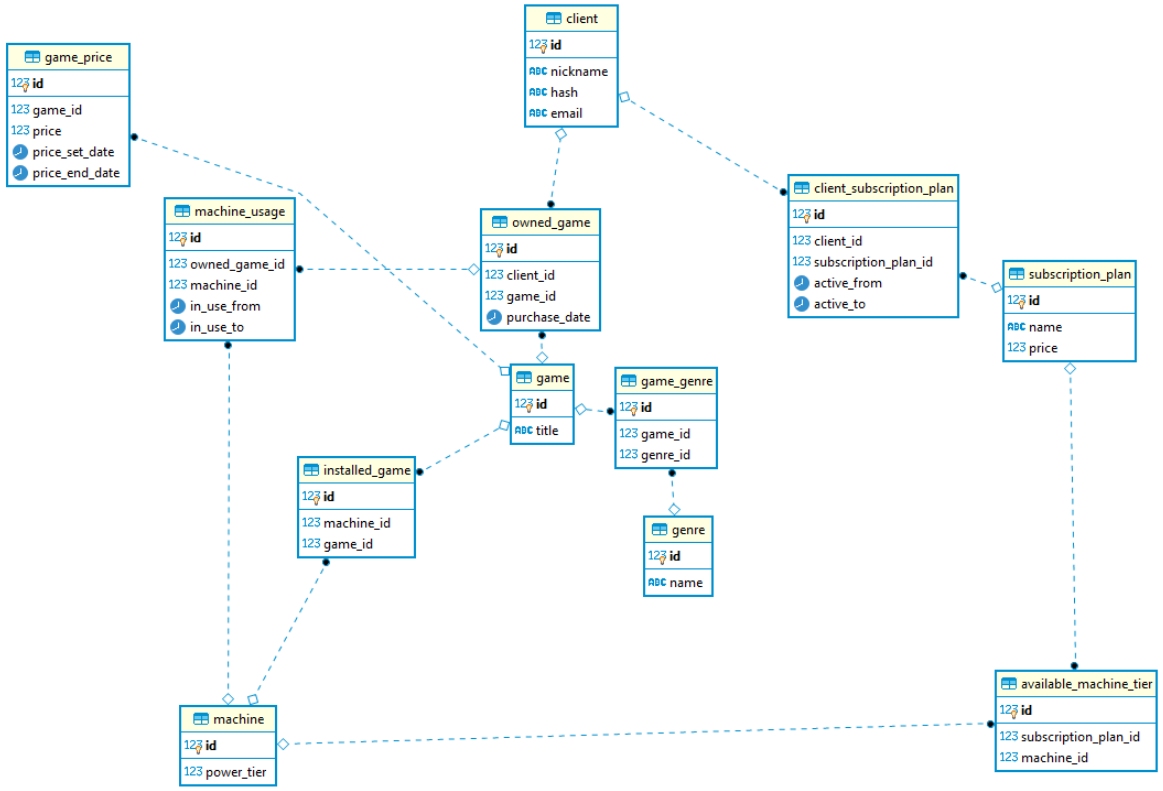

Рисунок 2.1. Схема базы данных

2.2. Возможности API

Возможности API можно разделить на две категории:

- Доступные всем
  - Вход и регистрация;
  - Выдача списка игр, отсортированного по разным параметрам;
  - Выдача списка жанров;
  - Выдача существующих планов по подписке.
- Доступные только авторизованным пользователям
  - Изменения данных аккаунта;
  - Выдача списка приобретенных игр;
  - Возможность покупки игр;
  - Выдача списка действующих и истекших клиентских планов по подписке;
  - Возможность покупки плана по подписке;
  - Возможность симуляции проведения игровой сессии;
  - Выдача информации о предыдущих игровых сессиях.

3. Сервер

Полный код сервера находится по ссылке `http://gitlab.icc.spbstu.ru/kalashnikov.ra/videogame-streaming-service-API`.

3.1. Связь с базой данных

Данные для подключения к базе данных хранятся в файле переменных среды. Код, представленный в листинге 1, используется для конфигурации подключения к ней.

Листинг 1: Конфигурация подключения к базе данных

```
import { Pool } from 'pg';

import env from '../../../env'

const databaseConfig = { connectionString: env.database_url };
const pool = new Pool(databaseConfig);

export default pool;
```

Код из листинга 2 отвечает за подключение к базе.

Листинг 2: Подключения к базе данных

```
import pool from './pool';

pool.on('connect', () => {
  console.log('connection established');
});

pool.on('remove', () => {
  console.log('client removed');
  process.exit(0);
});

require('make-runnable');
```

Для отправки запросов базе данных используется функция, показанная в листинге 3. На вход ей передается не только текст запроса, но и параметры.

Листинг 3: Отправление запросов базе данных

```
import pool from './pool';

export default {
  query(quertText, params) {
    return new Promise((resolve, reject) => {
      pool.query(quertText, params)
        .then((res) => {
          resolve(res);
        })
        .catch((err) => {
          reject(err);
        });
    });
  },
};
```

3.2. Вспомогательные функции

В качестве ответа сервер посылает сообщение, в поле *status* которого указывается результат, а также код.

Листинг 4: Сообщения о результате

```
const successMessage = { status: 'success' };
```

```
const errorMessage = { status: 'error' };
const status = {
  success: 200,
  error: 500,
  notfound: 404,
  unauthorized: 401,
  conflict: 409,
  created: 201,
  bad: 400,
  nocontent: 204,
};

export {
  successMessage,
  errorMessage,
  status,
};
```

Для хэширования пароля используется библиотека *bcrypt*, а для токенов библиотека *jsonwebtoken*. В листинге 5 представлен файл с вспомогательными функциями, использующимися для генерации и сравнения паролей, а также для создания токена, который генерируется на основе id, почтового адреса и никнейма пользователя.

Листинг 5: Вспомогательные функции

```
import env from '../../env';
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';

const saltRounds = 10;
const salt = bcrypt.genSaltSync(saltRounds);
const hashPassword = password => bcrypt.hashSync(password, salt);


const generateUserToken = (id, email, nickname) => {
  const token = jwt.sign({
    id,
    email,
    nickname
  },
  env.secret, {expiresIn: '3d'});
  return token
}

const comparePassword = (hashedPassword, password) => {
  return bcrypt.compareSync(password, hashedPassword);
};

export {
  hashPassword,
  generateUserToken,
  comparePassword,
};
```

3.3. Авторизация

Верификация пользователя происходит по токену, сгенерированному на основе id пользователя в базе данных, почтового адреса, а также никнейма. Для этого используется

функция *verifyToken*, показанная в листинге 6. Она указывается первой в списке выполняющихся подряд функций и проверяет, что приложенный к запросу токен действительно сгенерирован данным сервером, после чего декодирует из него данные пользователя, а затем проверяет, что пользователь с такими данными действительно существует в базе данных. Если все хорошо, после этого управление передается следующей функции, которая, благодаря декодированным данным пользователя, может персонализировать дальнейшие запросы.

Листинг 6: Проверка токена

```
import jwt from 'jsonwebtoken';
import dotenv from 'dotenv';
import {
  errorMessage, status,
} from '../helpers/status';
import dbQuery from '../db/dev/dbQuery';


dotenv.config();

const verifyToken = async (req, res, next) => {
  const token = req.get('token');
  if (!token) {
    errorMessage.error = 'Token not provided';
    return res.status(status.bad).send(errorMessage);
  }
  try {
    const decoded = jwt.verify(token, process.env.SECRET);
    req.user = {
      email: decoded.email,
      id: decoded.id,
      nickname: decoded.nickname
    };
    const checkUserQuery = 'SELECT * FROM client WHERE id = $1 AND email = $2 AND nickname = $3';
    try {
      const { rows } = await dbQuery.query(checkUserQuery, [req.user.id, req.user.email, req.user.nickname]);
      const dbResponse = rows[0];

      if (!dbResponse) {
        errorMessage.error = 'Authentification failed';
        return res.status(status.notfound).send(errorMessage);
      }
    } catch (error) {
      console.log(`${error}`);
      errorMessage.error = 'Error occured while trying to verify token';
      return res.status(status.error).send(errorMessage);
    }
    next();
  } catch (error) {
    errorMessage.error = 'Authentication Failed';
    return res.status(status.unauthorized).send(errorMessage);
  }
};

export default verifyToken;
```

Для обработки запросов, связанных с данными пользователя, используется 3 функции: *createUser*, *signinUser* и *updateUser*. Первая из них представлена в листинге 7. Она

отвечает за создание нового пользователя. Данные пользователя, включающие почтовый адрес, пароль и никнейм, передаются в теле запроса. Сначала производится проверка валидности данных, затем производится попытка добавить пользователя в базу данных, перед которой пароль хэшируется. Если попытка добавления завершается с ошибкой, сообщение об этом отправляется пользователю, иначе происходит генерация токена и отправка успешного сообщения пользователю с этим токеном и данными, на основе которых он создан.

Листинг 7: Создание нового пользователя

```
const createUser = async (req, res) => {
  if (req.body.email === undefined || req.body.password === undefined || req.
   ↪ body.nickname === undefined) {
    errorMessage.error = 'Email, password or nickname field are not provided';
    return res.status(status.bad).send(errorMessage);
  }

  const {
    email, password, nickname,
  } = req.body;


  if (validator.isEmpty(email) || validator.isEmpty(password) || validator.
   ↪ isEmpty(nickname)) {
    errorMessage.error = 'Email, password and nickname field cannot be empty';
    return res.status(status.bad).send(errorMessage);
  }

  if (!validator.isEmail(email)) {
    errorMessage.error = 'Email is not valid';
    return res.status(status.bad).send(errorMessage);
  }

  if (!validator.isLength(password, 5)) {
    errorMessage.error = 'Password must be not shorter than 5 characters';
    return res.status(status.bad).send(errorMessage);
  }

  const hashedPassword = hashPassword(password);
  const createUserQuery = `INSERT INTO
      client(email, nickname, hash)
     VALUES($1, $2, $3)
      returning *`;
  const values = [
    email,
    nickname,
    hashedPassword
  ];

  try {
    const { rows } = await dbQuery.query(createUserQuery, values);
    const dbResponse = rows[0];
    delete dbResponse.hash;
    const token = generateUserToken(dbResponse.id, dbResponse.email, dbResponse.
   ↪ nickname);
    successMessage.data = dbResponse;
    successMessage.data.token = token;
    return res.status(status.created).send(successMessage);
  } catch (error) {
    if (error.routine === '_bt_check_unique') {
```

```
48        errorMessage.error = 'User␣with␣that␣EMAIL␣already␣exist';
```

Функция входа пользователя *signinUser* отображена в листинге 8. Данная функция проверят данные, переданные в теле запроса на соответствие ограничениям, а также на их верность путем сравнения с данными, содержащимися в базе. В случае если проверка проходит успешно, в отправленное в ответ сообщение помещается токен, который клиент сможет использовать для дальнейших запросов, иначе в ответ отправляется сообщение об ошибке с пояснением проблемы.

Листинг 8: Логин пользователя

```
1     }
2     errorMessage.error = 'Error␣occured␣while␣trying␣to␣create␣user';
3     return res.status(status.error).send(errorMessage);
4   }
5 }
6
7 const signinUser = async (req, res) => {
8   const { email, password } = req.body;
9
10  if (validator.isEmpty(email) || validator.isEmpty(password)) {
11    errorMessage.error = 'Email␣or␣Password␣detail␣is␣missing';
12    return res.status(status.bad).send(errorMessage);
13  }
14
15  if (!validator.isEmail(email) || !validator.isLength(password, 5)) {
16    errorMessage.error = 'Please␣enter␣a␣valid␣Email␣or␣Password';
17    return res.status(status.bad).send(errorMessage);
18  }
19
20  const signinUserQuery = 'SELECT␣*␣FROM␣client␣WHERE␣email␣=␣$1';
21  try {
22    const { rows } = await dbQuery.query(signinUserQuery, [email]);
23    const dbResponse = rows[0];
24
25    if (!dbResponse) {
26      errorMessage.error = 'User␣with␣this␣email␣does␣not␣exist';
27      return res.status(status.notfound).send(errorMessage);
28    }
29
30    if (comparePassword(password, dbResponse.hash)) {
31      errorMessage.error = 'The␣password␣you␣provided␣is␣incorrect';
32      return res.status(status.bad).send(errorMessage);
33    }
34    const token = generateUserToken(dbResponse.id, dbResponse.email, dbResponse.
    ↪ nickname);
35    delete dbResponse.hash;
36    successMessage.data = dbResponse;
37    successMessage.data.token = token;
```

Пример сообщения, отправляемого двумя предыдущими запросами, представлен в листинге 9.

Листинг 9: Пример сообщения при попытке создания нового пользователя

```
1 {
2     "status": "success",
3     "data": {
4         "id": 5024,
5         "nickname": "sampleuser",
```

```
6           "email": "sample_1@mail.ru",
7           "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6
    ↪ NTAyNCwiZW1haWwiOiJzYW1wbGVfMUBtYWlsLnJ1Iiwibmlja25
    ↪ hbWUiOiJzYW1wbGV1c2VyIiwiaWF0IjoxNTkyMzM5MjU1LCJleHAiOjE1
    ↪ OTI1OTg0NTV9.RMTmZ87C_Ee8x4BJA9zp_ZkwCrfv9b--6rojrBcrSbw"
8       }
9   }
```

В листинге 10 показана функция *updateUser*, отвечающая за обновление данных пользователя. В теле запроса должны быть или пароль, или никнейм, или оба сразу, иначе функция вернет сообщение об ошибке. На основе того, какие аргументы были предоставлены, функция выбирает подходящий запрос в базу данных, после чего выполняет его. Сообщение, возвращаемое ей, аналогично таковому из листинга 9.

Листинг 10: Обновление информации пользователя

```
1   } catch (error) {
2     errorMessage.error = 'Error occured while trying to signin';
3     return res.status(status.error).send(errorMessage);
4   }
5 }
6
7 const updateUser = async (req, res) => {
8   if (req.body.password === undefined && req.body.nickname === undefined) {
9     errorMessage.error = 'Nothing to update';
10    return res.status(status.bad).send(errorMessage);
11  }
12
13  var updateUserQuery;
14  var values;
15  if (req.body.password !== undefined && req.body.nickname !== undefined) {
16    if (validator.isEmpty(req.body.password) || validator.isEmpty(req.body.
    ↪ nickname)) {
17      errorMessage.error = 'Password and nickname field cannot be empty';
18      return res.status(status.bad).send(errorMessage);
19    }
20    if (!validator.isLength(req.body.password, 5)) {
21      errorMessage.error = 'Password must be not shorter than five(5) characters
    ↪ ';
22      return res.status(status.bad).send(errorMessage);
23    }
24    const hashedPassword = hashPassword(req.body.password);
25    updateUserQuery = `UPDATE client SET nickname = $1, hash = $2 WHERE id = $3
    ↪ returning *`;
26    values = [req.body.nickname, hashedPassword, req.user.id];
27  } else {
28    if (req.body.password !== undefined) {
29      if (validator.isEmpty(req.body.password)) {
30        errorMessage.error = 'Password field cannot be empty';
31        return res.status(status.bad).send(errorMessage);
32      }
33      if (!validator.isLength(req.body.password, 5)) {
34        errorMessage.error = 'Password must be not shorter than five(5) 
    ↪ characters';
35        return res.status(status.bad).send(errorMessage);
36      }
37      const hashedPassword = hashPassword(req.body.password);
38      updateUserQuery = `UPDATE client SET hash = $1 WHERE id = $2 returning *`;
39      values = [hashedPassword, req.user.id];
```

```
    } else {
      if (validator.isEmpty(req.body.nickname)) {
        errorMessage.error = 'Nickname field cannot be empty';
        return res.status(status.bad).send(errorMessage);
      }
      values = [req.body.nickname, req.user.id];
      updateUserQuery = `UPDATE client SET nickname = $1 WHERE id = $2 returning *`;
    }
  }

  try {
    const { rows } = await dbQuery.query(updateUserQuery, values);
    const dbResponse = rows[0];
    delete dbResponse.hash;
    const token = generateUserToken(dbResponse.id, dbResponse.email, dbResponse.nickname);
    successMessage.data = dbResponse;
    successMessage.data.token = token;
```

В листинге 11 показано расположение данных функций.

Листинг 11: Расположение функций

```
const router = express.Router();

router.post('/auth/signup', createUser);
router.post('/auth/signin', signinUser);
router.post('/users/me', verifyAuth, updateUser);
```

3.4. Обработка запросов

3.4.1. Игры

Для работы с информацией об играх на сервере созданы 5 функций: *getAllGames*, *getUserGames*, *getGenres*, *getGamesOfGenre* и *addUserGame*. Первая из них представлена в листинге 12. Данная функция выводит полный список игр, отсортированный в соответствии с переданными параметрами. Возможна сортировка по алфавиту, цене и популярности, которая считается по времени сыгранном в игру всеми пользователями, порядок также является параметром. В случае если никакие параметры указаны не были, будет выбрана сортировка по алфавиту по возрастанию. В функции сначала на основе параметров выбирается запрос, а затем он выполняется, и результат отправляется клиенту.

Листинг 12: Выборка всех игр

```
const getAllGames = async (req, res) => {
  var getAllGamesQuery
  if (req.query.sort === undefined) {
    getAllGamesQuery = 'SELECT game.id, game.title, game_price.price FROM game INNER JOIN game_price ON game.id = game_price.id AND game_price.price_end_date IS NULL ORDER BY title ASC';
  } else {
    if (req.query.sort == "title" || req.query.sort == "price") {
      if (req.query.order === undefined) {
        getAllGamesQuery = `SELECT game.id, game.title, game_price.price FROM game INNER JOIN game_price ON game.id = game_price.id AND game_price.price_end_date IS NULL ORDER BY ${req.query.sort} ASC`;
      } else {
```

```
10          getAllGamesQuery = `SELECT game.id, game.title, game_price.price FROM
    ↪ game INNER JOIN game_price ON game.id = game_price.id AND game_price.
    ↪ price_end_date IS NULL ORDER BY ${req.query.sort} ${req.query.order}`;
11        }
12      } else {
13        if (req.query.sort == "popularity") {
14          if (req.query.order === undefined) {
15            getAllGamesQuery = `SELECT game.title as title,game_price.price as
    ↪ price, SUM((date_part('hour', machine_usage.in_use_to) * 60 + date_part('
    ↪ minute', machine_usage.in_use_to)
16            - date_part('hour', machine_usage.in_use_from) * 60 - date_part('minute'
    ↪ , machine_usage.in_use_from))) as minutes_played
17          FROM owned_game
18          RIGHT JOIN machine_usage ON owned_game.id = machine_usage.owned_game_id
19          INNER JOIN game ON owned_game.game_id = game.id
20          INNER JOIN game_price ON owned_game.game_id = game_price.game_id AND
    ↪ game_price.price_end_date is NULL
21          GROUP BY title, price
22          ORDER BY minutes_played DESC`
23          } else {
24            getAllGamesQuery = `SELECT game.title as title,game_price.price as
    ↪ price, SUM((date_part('hour', machine_usage.in_use_to) * 60 + date_part('
    ↪ minute', machine_usage.in_use_to)
25            - date_part('hour', machine_usage.in_use_from) * 60 - date_part('minute'
    ↪ , machine_usage.in_use_from))) as minutes_played
26          FROM owned_game
27          RIGHT JOIN machine_usage ON owned_game.id = machine_usage.owned_game_id
28          INNER JOIN game ON owned_game.game_id = game.id
29          INNER JOIN game_price ON owned_game.game_id = game_price.game_id AND
    ↪ game_price.price_end_date is NULL
30          GROUP BY title, price
31          ORDER BY minutes_played ${req.query.order}`
32          }
33        } else {
34          getAllGamesQuery = 'SELECT␣game.id,␣game.title,␣game_price.price␣FROM␣
    ↪ game␣INNER␣JOIN␣game_price␣ON␣game.id␣=␣game_price.id␣AND␣game_price.
    ↪ price_end_date␣IS␣NULL␣ORDER␣BY␣title␣ASC';
35          }
36        }
37    }
38    try {
39      const { rows } = await dbQuery.query(getAllGamesQuery);
40      const dbResponse = rows;
41      if (dbResponse[0] === undefined) {
42        errorMessage.error = 'There␣are␣no␣games';
43        return res.status(status.notfound).send(errorMessage);
44      }
45      successMessage.data = dbResponse;
46      return res.status(status.success).send(successMessage);
47    } catch (error) {
48      errorMessage.error = 'An␣error␣Occured';
49      return res.status(status.error).send(errorMessage);
50    }
51  }
```

В сообщении содержится информация об играх: id в базе, цена, название, а также общее наигранное время всех пользователей, если сортировка была произведена по популярности. В листинге 13 показан фрагмент сообщения с сортировкой по цене по возрастанию.

Листинг 13: Пример сообщения при попытке получить все игры

```
{
    "status": "success",
    "data": [
        {
            "id": 3834,
            "title": "H0I7sX",
            "price": 200
        },
        {
            "id": 3650,
            "title": "vU8ntZYo",
            "price": 200
        },
        ...
        {
            "id": 4068,
            "title": "pPDKJD21Qh4t2TkZ",
            "price": 8000
        }
    ]
}
```

Следующая функция, *getUserGames*, показана в листинге 14. Данная функция возвращает игры, принадлежащие пользователю, поэтому к запросу должен быть приложен токен. Список игр, получаемый с помощью данной функции, также может быть отсортирован.

Листинг 14: Выборка игр пользователя

```
const getUserGames = async (req, res) => {
  var getUserGamesQuery
  if (req.query.sort === undefined) {
    getUserGamesQuery = 'SELECT game.title, owned_game.purchase_date FROM owned_game INNER JOIN game ON game_id = game.id WHERE client_id = $1 ORDER BY game.title ASC';
  } else {
    if (req.query.order === undefined) {
      getUserGamesQuery = `SELECT game.title , owned_game.purchase_date FROM owned_game INNER JOIN game ON game_id = game.id WHERE client_id = $1 ORDER BY ${req.query.sort} ASC`;
    } else {
      getUserGamesQuery = `SELECT game.title , owned_game.purchase_date FROM owned_game INNER JOIN game ON game_id = game.id WHERE client_id = $1 ORDER BY ${req.query.sort} ${req.query.order}`;
    }
  }
  const values = [req.user.id];
  try {
    const { rows } = await dbQuery.query(getUserGamesQuery, values);
    const dbResponse = rows;
    if (dbResponse[0] === undefined) {
      errorMessage.error = 'This user has no games';
      return res.status(status.notfound).send(errorMessage);
    }
    successMessage.data = dbResponse;
```

```
    return res.status(status.success).send(successMessage);
  } catch (error) {
    errorMessage.error = 'An error Occured while trying to get users games';
    return res.status(status.error).send(errorMessage);
  }
}
```

В ответном сообщении информация об играх несколько отличается от предыдущей функции, так как добавляется дата приобретения. Пример такого сообщения показан в листинге 15.

Листинг 15: Пример сообщения при попытке получить все игры пользователя

```
{
    "status": "success",
    "data": [
        {
            "title": "DOOM",
            "purchase_date": "2019-12-31T21:00:00.000Z"
        }
    ]
}
```

Функция *addUserGame*, показанная в листинге 16, отвечает за добавление игр пользователю. В качестве параметров ей должны быть переданы название игры, а также дата покупки. Данная функция также требует наличие токена в заголовке запроса. Помимо правильности формата введенных данных, данная функция проверяет нет ли уже у пользователя игры, которую ему пытаются добавить. Если добавление невозможно, сервер отправляет сообщение с ошибкой.

Листинг 16: Добавление игр пользователю

```
const addUserGame = async (req, res) => {
  if (req.body.game_title === undefined) {
    errorMessage.error = 'Title of the game is not specified';
    return res.status(status.error).send(errorMessage);
  }
  if (req.body.purchase_date === undefined) {
    errorMessage.error = 'Purchase date is not specified';
    return res.status(status.error).send(errorMessage);
  }
  if (!validator.isISO8601(req.body.purchase_date)) {
    errorMessage.error = 'Date is in wrong format, use ISO8601';
    return res.status(status.error).send(errorMessage);
  }
  const game_title = req.body.game_title;
  const purchase_date = req.body.purchase_date;
  const addUserGameQuery = 'INSERT INTO owned_game(client_id, game_id,
   ↪ purchase_date) VALUES ($1, (SELECT id FROM game WHERE title = $2), $3)';
  const getUsersGames = 'SELECT game.title FROM owned_game INNER JOIN game ON
   ↪ game_id = game.id WHERE client_id = $1';
  var values = [req.user.id]
  var ownedGames = []
  try {
    const { rows } = await dbQuery.query(getUsersGames, values);
    const dbResponse = rows;
    if (dbResponse[0] === undefined) {

    } else {
```

```
26        for (var i = 0; i < dbResponse.length; i++) {
27          ownedGames.push(dbResponse[i].title)
28        }
29      }
30    } catch (error) {
31      errorMessage.error = 'An error Occured while trying to get users games';
32      return res.status(status.error).send(errorMessage);
33    }
34    if (ownedGames.includes(game_title)) {
35      errorMessage.error = 'User already has this game';
36      return res.status(status.error).send(errorMessage);
37    }
38    values = [req.user.id, game_title, purchase_date]
39    try {
40      const { rows } = await dbQuery.query(addUserGameQuery, values);
41      const dbResponse = rows[0];
42      successMessage.data = dbResponse;
43      return res.status(status.created).send(successMessage);
44    } catch (error) {
45      console.log(`${error}`);
46      errorMessage.error = 'An error Occured while trying to add users game';
47      return res.status(status.error).send(errorMessage);
48    }
49  }
```

Ответное сообщение содержит лишь информацию об успешности операции. Пример показан в листинге 17.

Листинг 17: Пример сообщения при попытке добавить игру пользователю

```
1  {
2      "status": "success"
3  }
```

Следующая функция, *getGenres*, показана в листинге 18. Данная функция возвращает список жанров, существующих в базе.

Листинг 18: Выборка всех жанров

```
1  const getGenres = async (req, res) => {
2    const getGenresQuery = 'SELECT name FROM genre';
3    try {
4      const { rows } = await dbQuery.query(getGenresQuery);
5      const dbResponse = rows;
6      if (dbResponse[0] === undefined) {
7        errorMessage.error = 'There are no genres';
8        return res.status(status.notfound).send(errorMessage);
9      }
10     successMessage.data = dbResponse;
11     return res.status(status.success).send(successMessage);
12   } catch (error) {
13     errorMessage.error = 'An error Occured while trying to get genres';
14     return res.status(status.error).send(errorMessage);
15   }
16 }
```

В листинге 19 показан пример ответного сообщения.

Листинг 19: Пример сообщения при выборке жанров

```
1  {
2      "status": "success",
```

```
    "data": [
        {
            "name": "action"
        },
        {
            "name": "arcade"
        },
        {
            "name": "rpg"
        }
    ]
}
```

Наконец, функция *getGamesOfGenre*, возвращающая игры указанного жанра, представлена в листинге 20. Результат данного запроса также может быть отсортирован.

Листинг 20: Выборка всех игр определенного жанра

```
const getGamesOfGenre = async (req, res) => {
  var getGamesOfGenreQuery
  if (req.query.sort === undefined) {
    getGamesOfGenreQuery = 'SELECT game.title, game_price.price FROM game_genre
    ↪ INNER JOIN game ON game_id = game.id INNER JOIN genre ON genre_id = genre.
    ↪ id LEFT JOIN game_price ON game_genre.game_id = game_price.game_id AND
    ↪ game_price.price_end_date IS NULL WHERE genre.name = $1 ORDER BY game.
    ↪ title ASC';
  } else {
    if (req.query.order === undefined) {
      getGamesOfGenreQuery = `SELECT game.title, game_price.price FROM
    ↪ game_genre INNER JOIN game ON game_id = game.id INNER JOIN genre ON
    ↪ genre_id = genre.id LEFT JOIN game_price ON game_genre.game_id =
    ↪ game_price.game_id AND game_price.price_end_date IS NULL WHERE genre.name
    ↪ = $1 ORDER BY ${req.query.sort} ASC`;
    } else {
      getGamesOfGenreQuery = `SELECT game.title, game_price.price FROM
    ↪ game_genre INNER JOIN game ON game_id = game.id INNER JOIN genre ON
    ↪ genre_id = genre.id LEFT JOIN game_price ON game_genre.game_id =
    ↪ game_price.game_id AND game_price.price_end_date IS NULL WHERE genre.name
    ↪ = $1 ORDER BY ${req.query.sort} ${req.query.order}`;
    }
  }
  const values = [req.params.genre];
  if (values[0] === undefined) {
    errorMessage.error = 'Genre is not specified';
    return res.status(status.error).send(errorMessage);
  }
  try {
    const { rows } = await dbQuery.query(getGamesOfGenreQuery, values);
    const dbResponse = rows;
    if (dbResponse[0] === undefined) {
      errorMessage.error = 'There are no games in this genre';
      return res.status(status.notfound).send(errorMessage);
    }
    successMessage.data = dbResponse;
    return res.status(status.success).send(successMessage);
  } catch (error) {
    errorMessage.error = 'An error Occured while trying to get users games';
    return res.status(status.error).send(errorMessage);
  }
```

```
30 }
```

Ответное сообщение по своей структуре соответствует сообщению из листинга 13. Размещение функций показано в листинге 21.

Листинг 21: Размещение функций, связанных с играми

```
const router = express.Router();

router.get('/games', getAllGames);
router.get('/games/my', verifyAuth, getUserGames);
router.post('/games/my', verifyAuth, addUserGame);
router.get('/games/genre/:genre', getGamesOfGenre);
router.get('/genres', getGenres);
```

3.4.2. Подписки

Для обработки запросов, связанных с информацией о подписках созданы 3 функции: *getAllSubPlans*, *getUserSubscriptionPlans* и *adduserSubscriptionPlan*. Первая из них показана в листинге 22. Данная функция возвращает все доступные планы подписки.

Листинг 22: Выборка всех доступных подписочных планов

```
const getAllSubPlans = async (req, res) => {
  const getAllSubPlansQuery = 'SELECT * FROM subscription_plan ORDER BY price';
  try {
    const { rows } = await dbQuery.query(getAllSubPlansQuery);
    const dbResponse = rows;
    if (dbResponse[0] === undefined) {
      errorMessage.error = 'There are no subscription plans';
      return res.status(status.notfound).send(errorMessage);
    }
    successMessage.data = dbResponse;
    return res.status(status.success).send(successMessage);
  } catch (error) {
    errorMessage.error = 'An error Occured while trying to get suscription plans
    ↪ ';
    return res.status(status.error).send(errorMessage);
  }
};
```

В ответное сообщение помещается информация о каждом плане подписки: название, цена. Пример сообщения представлен в листинге 23.

Листинг 23: Пример сообщения при попытке получения всех планов подписки

```
{
    "status": "success",
    "data": [
        {
            "id": 1,
            "name": "low-tier plan",
            "price": 100
        },
        {
            "id": 2,
            "name": "middle-tier plan",
            "price": 200
```

```
        },
        {
            "id": 3,
            "name": "high-tier plan",
            "price": 300
        }
    ]
}
```

Функция *getUserSubscriptionPlans* показана в листинге 24. Данная функция возвращает все клиентские планы по подписке, которые действуют или действовали когда-то. Функция требует наличие токена в заголовке запроса.

Листинг 24: Выборка всех подписочных планов пользователя

```
const getUserSubscriptionPlans = async(req, res) => {
  const getUserSubscriptionPlansQuery = 'SELECT subscription_plan.name, active_from, active_to FROM client_subscription_plan INNER JOIN subscription_plan ON subscription_plan_id = subscription_plan.id WHERE client_id = $1';
  const values = [req.user.id]
  try {
    const { rows } = await dbQuery.query(getUserSubscriptionPlansQuery, values);
    const dbResponse = rows;
    if (dbResponse[0] === undefined) {
      errorMessage.error = 'This user has no subscription plans';
      return res.status(status.notfound).send(errorMessage);
    }
    successMessage.data = dbResponse;
    return res.status(status.success).send(successMessage);
  } catch (error) {
    errorMessage.error = 'An error Occured while trying to get user suscription plans';
    return res.status(status.error).send(errorMessage);
  }
};
```

Ответное сообщение отличается от такового предыдущей функции. Теперь информация о плане также содержит даты его действия. Пример такого сообщения приведен в листинге 25.

Листинг 25: Пример сообщения при попытке получения всех планов подписки пользователя

```
{
    "status": "success",
    "data": [
        {
            "name": "low-tier plan",
            "active_from": "2019-12-31T21:00:00.000Z",
            "active_to": "2020-02-01T21:00:00.000Z"
        },
        {
            "name": "middle-tier plan",
            "active_from": "2020-04-30T21:00:00.000Z",
            "active_to": "2020-05-01T21:00:00.000Z"
        }
```

```
    ]
}
```

Последняя функция этой секции *addUserSubscriptionPlan* представлена в листинге 26. Данная функция отвечает за добавление плана пользователю. Параметрами функции являются название плана, а также даты начала и окончания его действия. Перед добавлением параметры проверяются на соответствие требованиям, после чего проверятся не имел ли данный пользователь данный план в промежуток времени, предоставленный в аргументах. Для корректной работы данная функция также требует наличие токена в заголовке запроса.

Листинг 26: Добавление подписочного плана пользователю

```
const addUserSubscriptionPlan = async(req, res) => {
  const addUserSubscriptionPlanQuery = 'INSERT INTO client_subscription_plan (
   ↪ client_id, subscription_plan_id, active_from, active_to) VALUES ($1, (
   ↪ SELECT id FROM subscription_plan WHERE name = $2), $3, $4)';
  const {
    plan_name, active_from, active_to
  } = req.body;
  if (plan_name === undefined || active_from === undefined || active_to ===
   ↪ undefined) {
    errorMessage.error = 'plan_name, active_from or active_to is missing';
    return res.status(status.error).send(errorMessage);
  }

  if(!validator.isISO8601(active_from) || !validator.isISO8601(active_to)) {
    errorMessage.error = 'Date format is wrong, use ISO8601';
    return res.status(status.error).send(errorMessage);
  }

  const userPlanPeriodQuery = 'SELECT active_from, active_to FROM
   ↪ client_subscription_plan WHERE client_id = $1 AND subscription_plan_id = (
   ↪ SELECT id FROM subscription_plan WHERE name = $2)'
  var dates = [[]];
  try {
    const { rows } = await dbQuery.query(userPlanPeriodQuery, [req.user.id,
   ↪ plan_name]);
    const dbResponse = rows;
    if (dbResponse[0] === undefined) {
    } else {
      for (var i =0; i < dbResponse.length; i++) {
        dates.push([dbResponse[i].active_from, dbResponse[i].active_to]);
      }
    }
  } catch (error) {
    errorMessage.error = 'An error Occured while trying to get user suscription
   ↪ plans';
    return res.status(status.error).send(errorMessage);
  }

  for (var i =0; i < dates.length; i++) {
    if ((validator.isBefore(active_from, format.asString(dates[i][1])) &&
   ↪ validator.isAfter(active_from, format.asString(dates[i][0]))) || (
   ↪ validator.isBefore(active_to, format.asString(dates[i][1])) && validator.
   ↪ isAfter(active_to, format.asString(dates[i][0])))) {
      errorMessage.error = 'User already has this plan for this period';
      return res.status(status.error).send(errorMessage);
    }
  }
```

```
38
39    const values = [req.user.id, plan_name, active_from, active_to];
40    try {
41      const { rows } = await dbQuery.query(addUserSubscriptionPlanQuery, values);
42      const dbResponse = rows[0];
43      successMessage.data = dbResponse;
44      return res.status(status.created).send(successMessage);
45    } catch (error) {
46          console.log(`${error}`);
47      errorMessage.error = 'An error Occured while trying to add users
      ↪ subscription plan';
48      return res.status(status.error).send(errorMessage);
49    }
50  };
```

Ответное сообщение данного запроса не несет дополнительной информации кроме статуса его выполнения.

Размещение функций показано в листинге 27.

Листинг 27: Размещение функций, связанных с подписочными планами

```
1  const router = express.Router();
2
3  router.get('/subscription_plans', getAllSubPlans);
4  router.get('/subscription_plans/my', verifyAuth, getUserSubscriptionPlans);
5  router.post('/subscription_plans/my', verifyAuth, addUserSubscriptionPlan);
```

3.4.3. Игровые сессии

Для управления игровыми сессиями созданы 2 функции: *getUserMachineUsage* и *createUserMachineUsage*. Данные функции связаны с выборкой и добавлением игровых сессий соответственно. Обе функции требуют наличие токена в заголовке запроса. Первая из них показана в листинге 28.

Листинг 28: Выборка всех сессий пользователя

```
1  const getUserMachineUsage = async (req, res) => {
2    const getUserMachineUsageQuery = 'SELECT game.title, in_use_from, in_use_to,
      ↪ machine_id FROM machine_usage LEFT JOIN owned_game ON owned_game_id =
      ↪ owned_game.id LEFT JOIN game ON owned_game.game_id = game.id WHERE
      ↪ owned_game.client_id = $1 ORDER BY in_use_from DESC';
3      const values = [req.user.id];
4      try {
5      const { rows } = await dbQuery.query(getUserMachineUsageQuery, values);
6      const dbResponse = rows;
7      if (dbResponse[0] === undefined) {
8        errorMessage.error = 'There is no machine usage';
9        return res.status(status.notfound).send(errorMessage);
10     }
11     successMessage.data = dbResponse;
12     return res.status(status.success).send(successMessage);
13   } catch (error) {
14     errorMessage.error = 'An error Occured while trying to get machine usage';
15     return res.status(status.error).send(errorMessage);
16   }
17 };
```

Ответное сообщение содержит информацию о сессиях, которая включает в себя: название игры, id использованной машины, а также время начала и завершения сессии. Пример сообщения показан в листинге 29.

Листинг 29: Пример сообщения при попытке выборки всех сессий пользователя

```
{
    "status": "success",
    "data": [
        {
            "title": "DOOM",
            "in_use_from": "2020-05-01T05:20:00.000Z",
            "in_use_to": "2020-05-01T09:22:01.000Z",
            "machine_id": "44"
        },
        {
            "title": "DOOM",
            "in_use_from": "2020-01-01T16:20:00.000Z",
            "in_use_to": "2020-01-01T17:22:01.000Z",
            "machine_id": "43"
        }
    ]
}
```

Функция *createUserMachineUsage* показана в листинге 30. В качестве параметров в эту функцию передаются название игры и два времени через точку с запятой: время начала игровой сессии и время ее завершения. Перед добавлением сессии в базу данных функция проверяет входные данные на соответствие требованиям, затем удостоверяется, что пользователь имеет игру, указанную в параметрах, после чего создает запись в базе.

Листинг 30: Добавление новой сессии

```
const createUserMachineUsage = async (req, res) => {
    const createUserMachineUsageQuery = `INSERT INTO machine_usage (owned_game_id, machine_id, in_use_from, in_use_to) VALUES ((SELECT id from owned_game WHERE client_id = $1 AND game_id = (SELECT id FROM game WHERE title = $2)), $3, $4, $5) returning *`;
    const usage_time = req.body.usage_time;
    if (usage_time === undefined) {
        errorMessage.error = 'Dates are missing';
    return res.status(status.error).send(errorMessage);
    }
    const dates = usage_time.split(";");
    if (dates[1] === undefined) {
        errorMessage.error = 'You need to specify both dates';
    return res.status(status.error).send(errorMessage);
    }
    if (!validator.isISO8601(dates[0]) || !validator.isISO8601(dates[1])) {
        errorMessage.error = 'Date format is wrong, use ISO8601';
    return res.status(status.error).send(errorMessage);
    }
    const game_title = req.body.game_title;
    if (game_title === undefined) {
        errorMessage.error = 'You need to specify game title';
    return res.status(status.error).send(errorMessage);
    }

    const checkIfUserHasGameQuery = 'SELECT * FROM owned_game WHERE client_id = $1 AND game_id = (SELECT id FROM game WHERE title = $2)';
    try {
        const { rows } = await dbQuery.query(checkIfUserHasGameQuery, [`${req.user.id}`, `${game_title}`]);
```

```
        const dbResponse = rows;
        if (dbResponse[0] === undefined) {
            errorMessage.error = 'User doesnt have this game';
            return res.status(status.bad).send(errorMessage);
        }
    } catch (error) {
        console.log(`${error}`);
        errorMessage.error = 'An error Occured while tring to check if user has game';
    return res.status(status.error).send(errorMessage);
    }

    const getFreeMachinesQuery = 'SELECT id FROM machine WHERE id NOT IN (SELECT machine_id FROM machine_usage WHERE in_use_from > $1 OR in_use_to < $2 GROUP BY machine_id)';
    var machineId = -1;
    try {
        const { rows } = await dbQuery.query(getFreeMachinesQuery, [`\'${dates[1]}\'`, `\'${dates[0]}\'`]);
        const dbResponse = rows;
        if (dbResponse[0] === undefined) {
            errorMessage.error = 'There is no free machines';
            return res.status(status.bad).send(errorMessage);
        }
        machineId = dbResponse[0].id;
    } catch (error) {
        errorMessage.error = 'An error Occured while tring to get free machines';
        return res.status(status.error).send(errorMessage);
    }

    const values = [req.user.id, game_title, machineId, dates[0], dates[1]];
    try {
        const { rows } = await dbQuery.query(createUserMachineUsageQuery, values);
        const dbResponse = rows[0];
        successMessage.data = dbResponse;
        return res.status(status.created).send(successMessage);
    } catch (error) {
        errorMessage.error = 'An error Occured while trying to add machine usage';
        return res.status(status.error).send(errorMessage);
    }
};
```

Ответное сообщение содержит информацию о сессии. Его пример показан в листинге 31.

Листинг 31: Пример сообщения при попытке добавления новой сессии

```
{
    "status": "success",
    "data": {
        "id": 5031,
        "owned_game_id": "5029",
        "machine_id": "43",
        "in_use_from": "2020-01-01T16:20:00.000Z",
        "in_use_to": "2020-01-01T17:22:01.000Z"
    }
}
```

Размещение функций показано в листинге 32.

Листинг 32: Размещение функций, связанных с игровыми сессиями

```
const router = express.Router();

router.post('/sessions/my', verifyAuth, createUserMachineUsage);
router.get('/sessions/my', verifyAuth, getUserMachineUsage);
```

3.5. Тестирование

Тестирование сервера было проведено путем отправления различных запросов и анализа полученных ответных сообщений. В данной секции будут приведены результаты тестирования функций с наибольшим количеством различных ответных сообщений: функции авторизации, а также добавления игровой сессии.

Начнем с тестирования функции *createUser*. Для начала проверим реакцию на отсутствие одного из полей в запросе. Запрос, а также ответное сообщение показаны в листингах 33 и 34.

Листинг 33: Тело запроса

```
{
  "email": "test@mail.ru",
  "password": "1234567"
}
```

Листинг 34: Тело ответа

```
{
    "status": "error",
    "error": "Email,
  ↪ password or nickname
  ↪  fields are not
  ↪ provided"
}
```

Теперь проверим как обрабатывается пустое поле, например пустой почтовый адрес.

Листинг 35: Тело запроса

```
{
  "email": "",
  "password": "1234567",
  "nickname": "nike"
}
```

Листинг 36: Тело ответа

```
{
    "status": "error",
    "error": "Email,
  ↪ password and
  ↪ nickname field
  ↪ cannot be empty"
}
```

Далее рассмотрим ситуацию, когда переданный почтовый адрес не является таковым по формату.

Листинг 37: Тело запроса

```
{
  "email": "totally not
   ↪ email",
  "password": "1234567",
  "nickname": "nike"
}
```

Листинг 38: Тело ответа

```
{
    "status": "error",
    "error": "Email is not
  ↪  valid"
}
```

Так как пароль не должен быть короче 5 символов, проверим как сервер отреагирует на меньший пароль

Листинг 39: Тело запроса

```
{
  "email": "test@mail.ru",
  "password": "12",
  "nickname": "nike"
}
```

Листинг 40: Тело ответа

```
{
    "status": "error",
    "error": "Password must be not shorter than 5 characters"
}
```

Теперь нужно проверить, что при передаче почтового адреса, на который уже зарегистрирован аккаунт возвращается соответствующее сообщение. Для этого сначала удостоверимся, что при хороших данных сервер работает верно.

Листинг 41: Тело запроса

```
{
  "email": "server_test@mail.ru",
  "password": "123456",
  "nickname": "nike"
}
```

Листинг 42: Тело ответа

```
{
    "status": "success",
    "data": {
        "id": 5028,
        "nickname": "nike",
        "email": "server_test@mail.ru",
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NTAyOCwiZW1haWwiOiJzZXJ2ZXJfdGVzdEBtYWlsLnJ1Iiwibmlja25hbWUiOiJuaWtlIiwiZXhwIjoxNTkyNzk0ODYzLCJleHAiOjE1OTMwNTQwNjN9.IP0J1618heAMLUm4e6tKxDs5DsqcPW1X7yRYR2vjdIk"
    }
}
```

Теперь попытаемся использовать тот же адрес еще раз.

Листинг 43: Тело запроса

```
{
  "email": "server_test@mail.ru",
  "password": "12345678",
  "nickname": "дnike"
}
```

Листинг 44: Тело ответа

```
{
    "status": "error",
    "error": "User with that EMAIL already exist"
}
```

Теперь проведем аналогичный тест функции *createUserMachineUsage*. Для начала выполним проверку обработки запроса без дат сессии.

Листинг 45: Тело запроса

```
{
  "game_title":"DOOM"
}
```

Листинг 46: Тело ответа

```
{
    "status": "error",
    "error": "Dates are
    ↪ missing"
}
```

Теперь проверим запрос с плохим форматом дат.В листингах 47 и 48 вместо двух дат указана лишь одна, а в листингах 49 и 50 указанные даты не соответствуют формату ISO8601.

Листинг 47: Тело запроса

```
{
  "usage_time":"2020-01-01
  ↪  19:20:20",
  "game_title":"DOOM"
}
```

Листинг 48: Тело ответа

```
{
    "status": "error",
    "error": "You need to
    ↪ specify both dates"
}
```

Листинг 49: Тело запроса

```
{
  "usage_time":"2020-01-01
  ↪  19:20:20;01-01-2020
  ↪  19:30:10",
  "game_title":"DOOM"
}
```

Листинг 50: Тело ответа

```
{
    "status": "error",
    "error": "Date format
    ↪ is wrong, use ISO860
    ↪ 1"
}
```

Далее проверим реакцию сервера при плохом названии игры в запросе. В листингах 51 и 52 указана игра, которой нет у пользователя.

Листинг 51: Тело запроса

```
{
  "usage_time":"2020-01-01
  ↪  19:20:20;2020-01-01
  ↪  19:30:10",
  "game_title":"0"
}
```

Листинг 52: Тело ответа

```
{
    "status": "error",
    "error": "User doesnt
    ↪ have this game"
}
```

Остальные функции были протестированы по аналогии с представленными выше.

4. Android приложение

Полный код приложения доступен по ссылке `http://gitlab.icc.spbstu.ru/kalashnikov.ra/videogame-streaming-service-adnroid-app`.

4.1. Взаимодействие с сервером

4.1.1. Настройка Retrofit

В интерфейсе *StreamingServiceAPI*, показанном в листинге 53, описываются функции сервера, рассмотренные в предыдущей секции, для дальнейшей работы с ними.

Листинг 53: Интерфейс для работы с сервером

```
interface StreamingServiceAPI {
    @POST("/auth/signup")
    fun signUp(@Body body: SignUpClient): Call<AuthorizationReport>

    @POST("/auth/signin")
    fun signIn(@Body body: SignInClient): Call<AuthorizationReport>

    @GET("/games")
    fun getGames(@Query("sort") sort: String?, @Query("order") order: String?):
    ↪ Call<GameReport>

    @GET("/games/my")
    fun getUserGames(@Header("token") token: String, @Query("sort") sort: String
    ↪ ?, @Query("order") order: String?): Call<GameReport>

    @POST("/games/my")
    fun addUserGame(@Header("token") token: String, @Body body: NewGame): Call<
    ↪ AddReport>

    @GET("/genres")
    fun getGenres(): Call<GetGenresReport>

    @GET("/games/genre/{chosen_genre}")
    fun getGamesOfGenre(
        @Path("chosen_genre") chosenGenre: String, @Query("sort") sort: String?,
    ↪  @Query(
            "order"
        ) order: String?
    ): Call<GameReport>

    @GET("/subscription_plans")
    fun getSubscriptionPlans(): Call<SubPlanReport>

    @GET("/subscription_plans/my")
    fun getUserSubscriptionPlans(@Header("token") token: String): Call<
    ↪ UserSubPlanReport>

    @POST("/subscription_plans/my")
    fun addUserSubscriptionPlan(@Header("token") token: String, @Body body:
    ↪ UserSubPlanBody): Call<AddReport>

    @GET("/sessions/my")
    fun getUserSessions(@Header("token") token: String): Call<
    ↪ GetMachineUsageReport>

    @POST("/sessions/my")
    fun addUserSession(@Header("token") token: String, @Body body: SessionData):
    ↪  Call<AddMachineUsageReport>

    @POST("/users/me")
    fun updateUser(@Header("token") token:String, @Body body: UpdateUserBody):
    ↪ Call<AuthorizationReport>
}
```

Для парсинга JSON-файлов созданы классы, представляющие собой модели различных ответных сообщений. В листинге 54 представлены модели сообщения, полученного в результате авторизации.

Листинг 54: Классы для парсинга JSON-файлов

```
class AuthorizationReport : Serializable {
    @SerializedName("status")
    @Expose
    var status: String? = null
    @SerializedName("data")
    @Expose
    var data: AuthorizationData? = null
}
class AuthorizationData : Serializable {
    @SerializedName("id")
    @Expose
    var id: Int? = null
    @SerializedName("nickname")
    @Expose
    var nickname: String? = null
    @SerializedName("email")
    @Expose
    var email: String? = null
    @SerializedName("token")
    @Expose
    var token: String? = null
}
```

Для создания тел запросов также созданы классы, представляющие их. Примеры для авторизации приведены в листинге 55.

Листинг 55: Классы для создания тела запроса авторизации

```
class SignUpClient(val email: String, val password: String, val nickname: String
    ↪ )

class SignInClient(val email: String, val password: String)
```

4.1.2. Сервис

Все общение с сервером происходит в рамках сервиса *StreamingService*, с которым активности взаимодействуют посредством мессенджера. Внутри сервера происходит настройка подключения к серверу, а также посыл и прием запросов серверу. В сервисе определен набор кодов, соответствующих различным запросам на сервер. Этот набор показан в листинге 56.

Листинг 56: Коды запросов

```
    companion object {
        const val SIGN_IN_QUERY = 1
        const val SIGN_UP_QUERY = 2
        const val GET_ALL_GAMES_QUERY = 3
        const val GET_USER_GAMES_QUERY = 4
        const val ADD_USER_GAME_QUERY = 5
        const val GET_GENRES_QUERY = 6
        const val GET_GAMES_OF_GENRE_QUERY = 7
        const val GET_ALL_SUB_PLANS_QUERY = 8
        const val GET_USER_SUB_PLANS_QUERY = 9
        const val ADD_USER_SUB_PLAN_QUERY = 10
        const val GET_USER_SESSIONS_QUERY = 11
        const val ADD_USER_SESSION_QUERY = 12
        const val UPDATE_USER_QUERY = 13
    }
```

На основе кода запроса сервис вызывает нужную функцию обработки. Структурно все функции обработки запроса похожи - каждая из них достает из сообщения нужные параметры и, если с этим не возникает проблем, посылает запрос на сервер. Внутри этой функции определяется обработчик ответа, который достает данные из ответа сервера, после чего посылает их в активность. В качестве примера в листинге 57 приводится функция, обрабатывающая запрос на добавление игры пользователю.

Листинг 57: Добавление игры пользователя

```
        private fun addUserGame(msg: Message) {
            val serviceAPI = mRetrofit.create(StreamingServiceAPI::class.java)
            val replyTo = msg.replyTo
            val token = msg.data.getString("token")

            val title = msg.data.getString("title")
            val purchaseDate = msg.data.getString("purchase_date")

            if (token != null && title != null && purchaseDate != null) {
                val call = serviceAPI.addUserGame(
                    token,
                    NewGame(title, purchaseDate)
                )

                call.enqueue(object : retrofit2.Callback<AddReport> {
                    override fun onFailure(call: Call<AddReport>, t: Throwable)
    ↪ {
                        Log.i("StreamingService", "call failed")
                    }

                    override fun onResponse(
                        call: Call<AddReport>,
                        response: Response<AddReport>
                    ) {
                        val msgWithClient = Message.obtain(
                            null,
                            ADD_USER_GAME_QUERY
                        )
                        val b = Bundle()
                        if (response.body() != null) {
                            b.putSerializable("data", response.body())
                        } else {
                            b.putString("error", response.errorBody().toString()
    ↪ )
                        }
                        msgWithClient.data = b
                        replyTo.send(msgWithClient)
                    }
                })
            }
        }
```

В листинге 58 показан пример работы с сервисом. Сначала сервис создается и привязывается к активности, затем создается класс, ответственный за обработку сообщений от сервиса, после чего в листинге 59 сервису отправляется сообщение.

Листинг 58: Привязка сервиса

```
    var mService: Messenger? = null
    private var mConnection = object : ServiceConnection {
        override fun onServiceConnected(className: ComponentName, service:
    ↪ IBinder) {
```

```
            mService = Messenger(service)
            mBound = true
        }

        override fun onServiceDisconnected(className: ComponentName) {
            mService = null
            mBound = false
        }
    }
    class ResponseHandler : Handler() {
        override fun handleMessage(msg: Message) {
            if (msg.what == StreamingService.SIGN_IN_QUERY || msg.what ==
    ↪ StreamingService.SIGN_UP_QUERY) {
                handleSign(msg)
            }
        }

        private fun handleSign(msg: Message) {
            if (msg.data.getSerializable("data") != null) {
                val authorizationReport = msg.data.getSerializable("data") as
    ↪ AuthorizationReport
                if (authorizationReport.status == "success") {
                    val authorizationData = authorizationReport.data as
    ↪ AuthorizationData
                    val editor = mainAct.pref.edit()
                    editor.putString(APP_PREFERENCES_TOKEN, authorizationData.
    ↪ token)
                    editor.putString(APP_PREFERENCES_EMAIL, authorizationData.
    ↪ email)
                    editor.putString(APP_PREFERENCES_NICKNAME, authorizationData
    ↪ .nickname)
                    editor.apply()
                    mainAct.toNextActivity()
                } else {
                    Toast.makeText(
                        mainAct.applicationContext,
                        "Something␣went␣wrong",
                        Toast.LENGTH_LONG
                    )
                        .show()
                }
            } else {
                Toast.makeText(mainAct, "Error␣occured", Toast.LENGTH_SHORT).
    ↪ show()
            }
        }

    }
```

Листинг 59: Отправка сообщения сервису

```
    private fun loginPressed() {
        val msg = Message.obtain(null,
            StreamingService.SIGN_IN_QUERY
        )
        val b = Bundle()
        b.putString("email", binding.emailText.text.toString())
        b.putString("password", binding.passwordText.text.toString())
        msg.data = b

        msg.replyTo = Messenger(MainActivity.ResponseHandler())
```

```
11          try {
12              (activity as MainActivity).mService?.send(msg)
13          } catch (e: RemoteException) {
14              e.printStackTrace()
15          }
16      }
```

4.2. Авторизация

При первом запуске приложения появляется экран с возможностью зарегистрироваться или войти в аккаунт, варианты его внешнего вида показаны на рисунках 4.1-4.2.

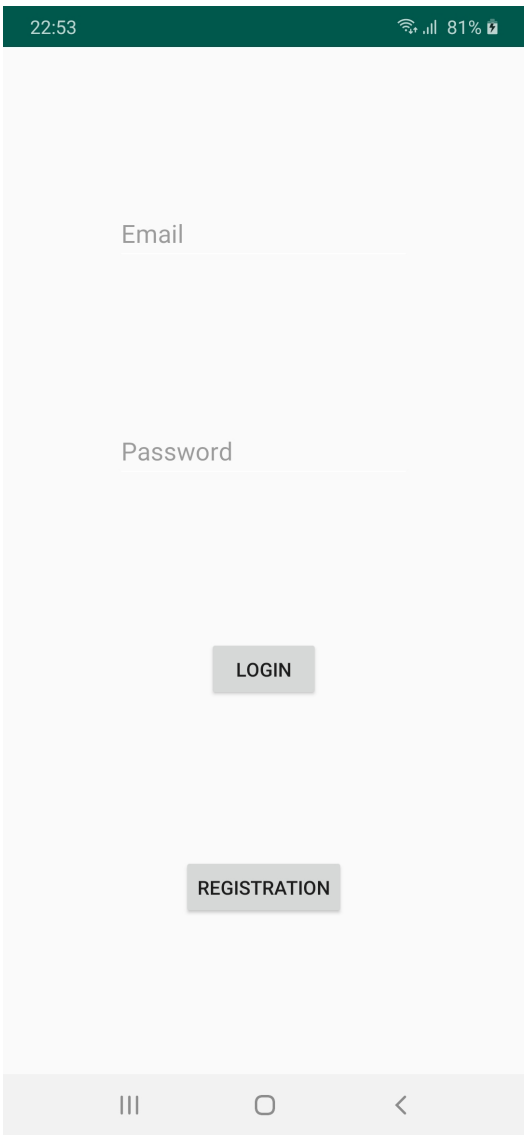

Рисунок 4.1. Экран входа

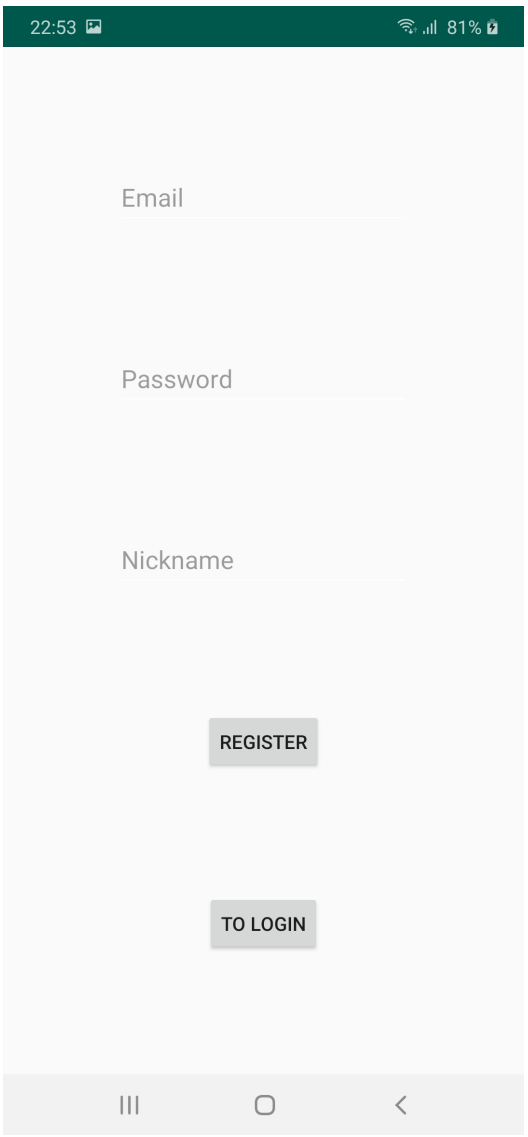

Рисунок 4.2. Экран регистрации

В случае если регистрация или вход проходит успешно, с помощью $sharedPreferences$ токен, почтовый адрес и никнейм сохраняются в память устройства и при следующем запуске приложения повторный вход не является необходимым. Код, осуществляющий проверку необходимости показа экрана входа находится в активности, имеющей тему NO_DISPLAY.

Навигация в приложении происходит с использованием $NavigationDrawer$, в заголовке которого отображается информация о пользователе(рисунок 4.3). В качестве плей-

схолдеров для картинок генерируются круглые картинки с буквой посередине. Для профиля пользователя выбирается первая буква его никнейма.

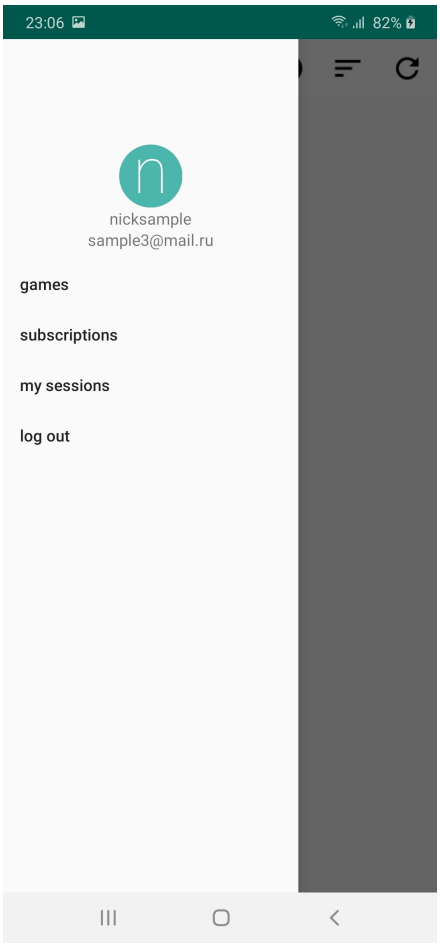

Рисунок 4.3. Navigation Drawer приложения

При нажатии на картинку профиля открывается диалог, позволяющий сменить пароль или никнейм, или и то и другое. При успешной смене необходимо повторно войти в приложение. Внешний вид диалога показан на рисунке 4.4.

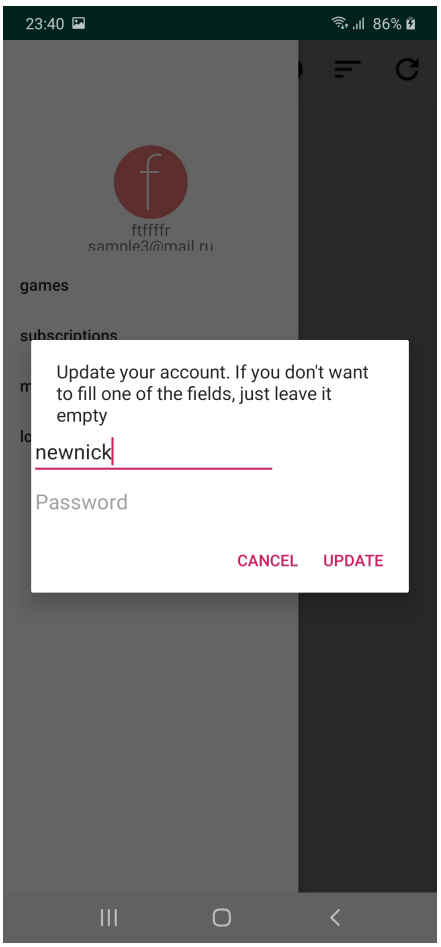

Рисунок 4.4. Обновление данных пользователя

4.3. Управление играми

Страница с играми содержит 2 фрагмента: фрагмент со всеми играми, где присутствует возможность их покупки, а также фрагмент, где отображаются игры пользователя, откуда можно произвести запуск игры. Переключение между фрагментами происходит с помощью кнопки на верхней панели. Для переключения на все игры нужно нажать иконку тележки, а для перехода к играм пользователя - иконку стрелочки. Также на верхней панели присутствует *spinner* с жанрами, который позволяет сортировать игры по жанрам, кнопка меняющая порядок сортировки и кнопка обновления страницы.

На рисунке 4.5 показан внешний вид списка игр, а на рисунке 4.6 показан список игр только одного жанра.

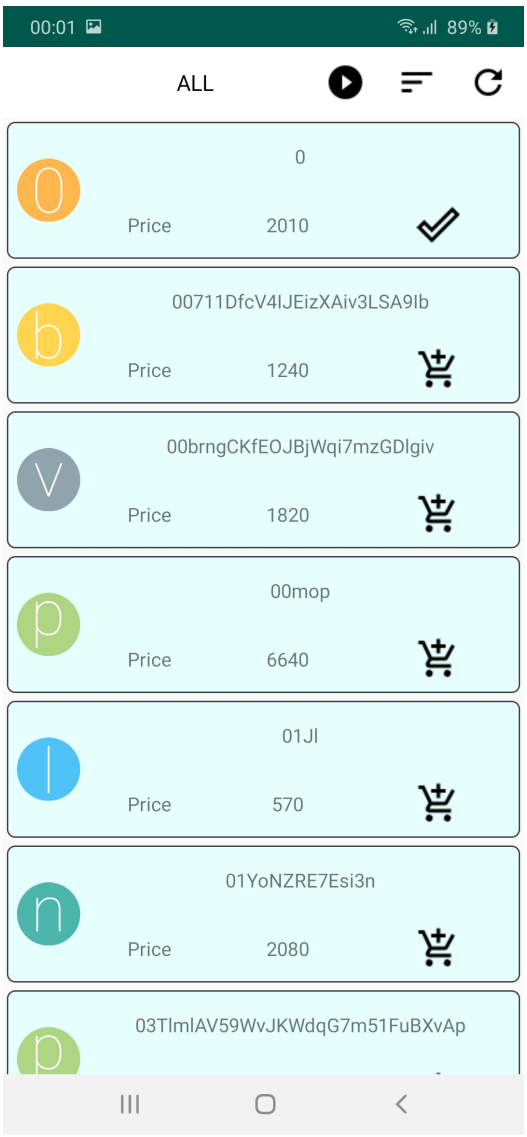

Рисунок 4.5. Все игры

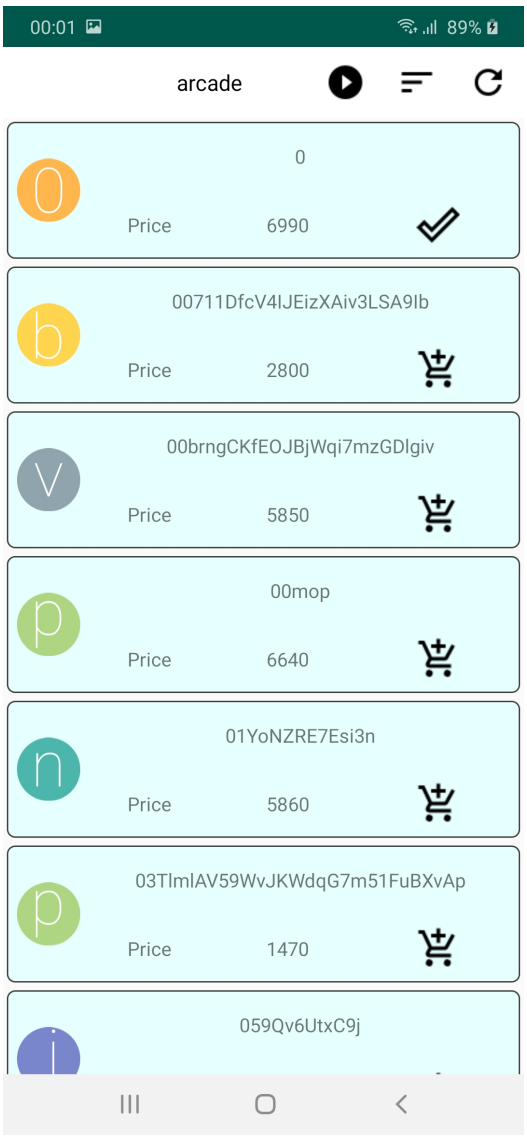

Рисунок 4.6. Игры конкретного жанра

При нажатии на иконку тележки с плюсом возникает диалог о покупке игры, представленный на рисунке 4.7. Список приобретенных игр показан на рисунке 4.8.

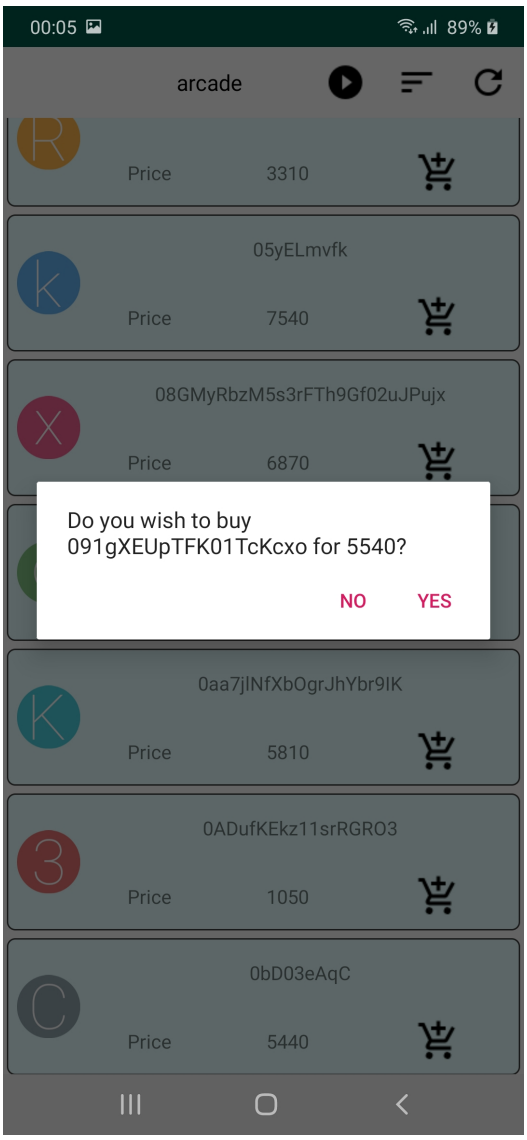

Рисунок 4.7. Диалог покупки игры

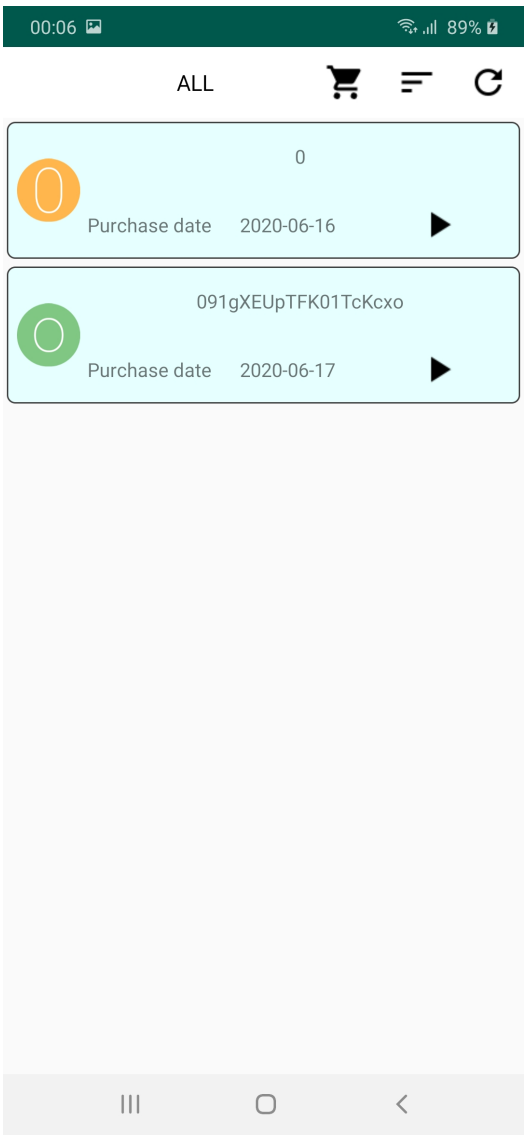

Рисунок 4.8. Игры пользователя

Для отображения списка игр в этой секции, а также списков подписок и сессий в последующих секциях используются адаптеры. В качестве примера в листинге 60 приведен адаптер списка всех игр. В качестве данных он требует список всех игр, а также список игр, приобретенных пользователем, чтобы отобразить возможность покупки игры. Картинка игры генерируется тем же методом, что и аватар пользователя, но так как в большинстве случаев игры будут отсортированы по алфавиту для генерации используется не первая буква названия, а последняя.

Листинг 60: Адаптер для списка всех игр

```
class AllGamesViewAdapter :
    RecyclerView.Adapter<AllGamesViewAdapter.ViewHolder>() {
    var data: List<Game> = ArrayList()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    var ownedGames: List<Game> = ArrayList()
        set(value) {
            field = value
            notifyDataSetChanged()
```

```
13          }
14
15      override fun getItemCount() = data.size
16
17      override fun onBindViewHolder(holder: ViewHolder, position: Int) {
18          val item = data[position]
19
20          holder.gameAvatar.setImageBitmap(
21              Utility().generateCircleBitmap(
22                  ManageActivity.manageAct,
23                  50.0f,
24                  item.title?.last().toString()
25              )
26          )
27          holder.gameTitle.text = item.title
28          holder.gamePrice.text = item.price.toString()
29          if (!ownedGames.contains(item)) {
30              holder.isBoughtImage.setImageResource(R.drawable.buy_game)
31              holder.isBoughtImage.setOnClickListener {
32
33                  val builder = AlertDialog.Builder(it.context)
34                  builder.setMessage("Do you wish to buy ${item.title} for ${item.
    ↪ price}?")
35                      .setPositiveButton("Yes") { dialog, _ ->
36                          val msg = Message.obtain(null, StreamingService.
    ↪ ADD_USER_GAME_QUERY)
37                          msg.replyTo = Messenger(AllGamesFragment.ResponseHandler
    ↪ ())
38                          val sdf = SimpleDateFormat("yyyy-MM-dd", Locale.US)
39                          val currentDate = sdf.format(Date())
40                          val b = Bundle()
41                          b.putString("title", item.title)
42                          b.putString("purchase_date", currentDate)
43                          b.putString(
44                              "token",
45                              ManageActivity.manageAct.pref.getString(
46                                  MainActivity.APP_PREFERENCES_TOKEN,
47                                  ""
48                              )
49                          )
50                          msg.data = b
51                          try {
52                              ManageActivity.manageAct.mService?.send(msg)
53                          } catch (e: RemoteException) {
54                              e.printStackTrace()
55                          }
56                          dialog.cancel()
57                      }
58                      .setNegativeButton("No") { dialog, _ ->
59                          dialog.cancel()
60                      }.show()
61              }
62          } else {
63              holder.isBoughtImage.setImageResource(R.drawable.game_bought)
64          }
65      }
66
67
68      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
    ↪ ViewHolder {
```

```
69          val layoutInflater = LayoutInflater.from(parent.context)
70          val view = layoutInflater.inflate(R.layout.games_list_item, parent,
        ↪ false)
71          return ViewHolder(view)
72      }
73
74      class ViewHolder constructor(view: View) : RecyclerView.ViewHolder(view) {
75          val gameAvatar: ImageView = view.gameAvatar
76          val gameTitle: TextView = view.gameTitle
77          val gamePrice: TextView = view.gamePrice
78          val isBoughtImage: ImageView = view.isBoughtImage
79      }
80  }
```

4.4. Управление подписками

Экран управления подписками имеет сходства с экраном управления играми. Здесь также имеется 2 фрагмента: список доступных планов по подписке, а также список пользовательских планах по подписке - активных и истекших. Переключение между ними осуществляется также, как и в списке игр. При нажатии на план по подписке на первом фрагменте откроется диалог, в котором можно выбрать длительность приобретаемого плана в месяцах. На рисунке 4.9 показан список доступных планов, а на рисунке 4.10 показан диалог покупки плана.

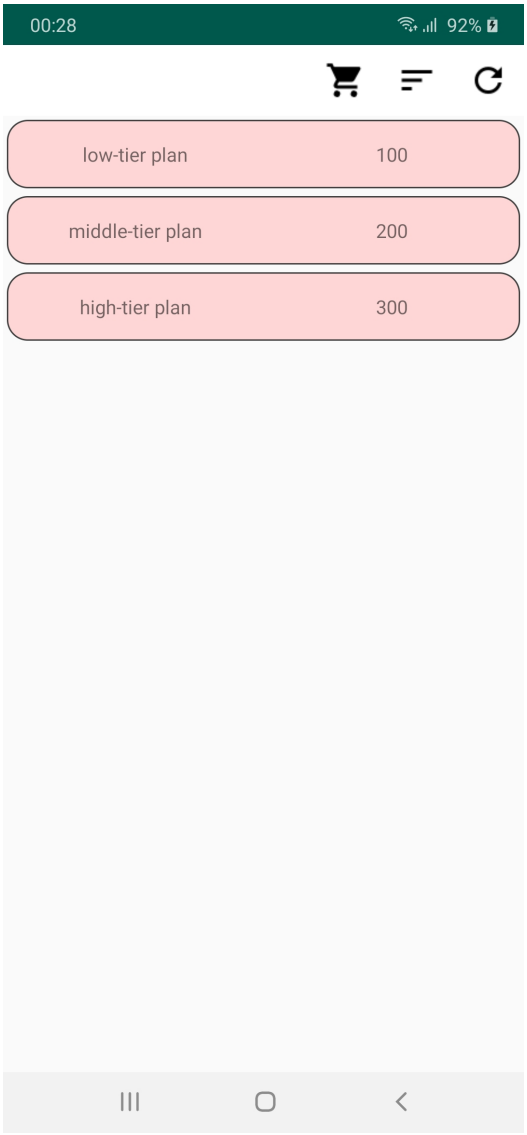

Рисунок 4.9. Список доступных планов

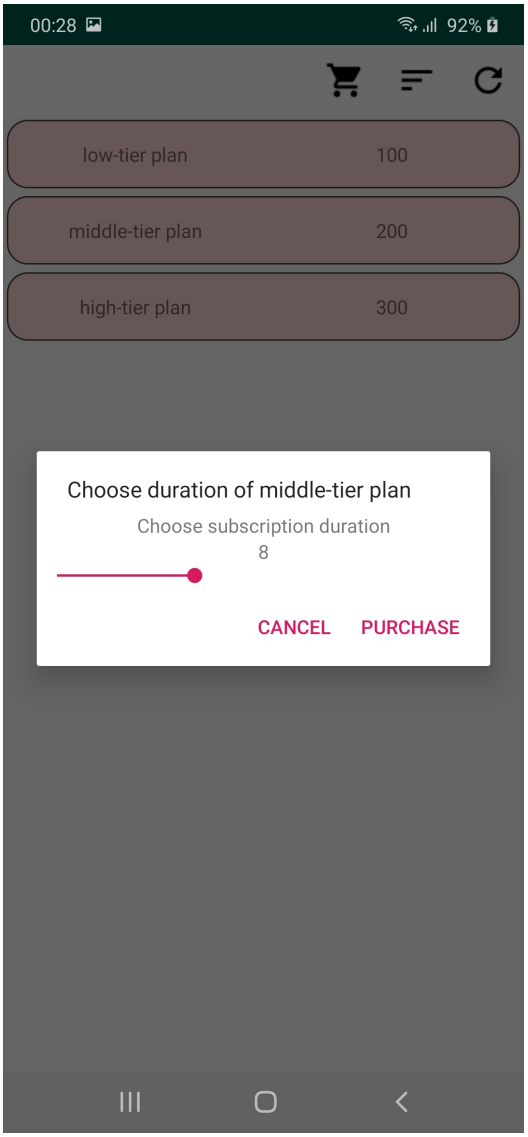

Рисунок 4.10. Диалог покупки плана

На рисунке 4.11 показан внешний вид списка пользовательских планов.

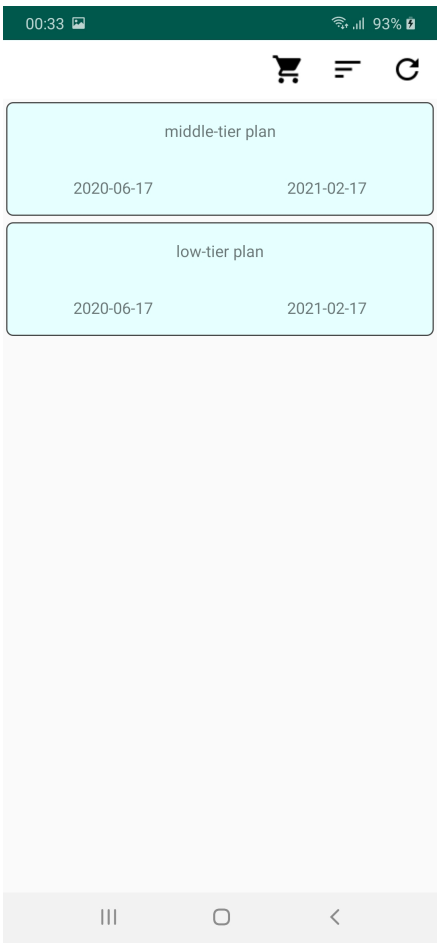

Рисунок 4.11. Планы подписки пользователя

4.5. Управление сессиями

Запустить игровую сессию можно нажав на иконку стрелки около названия игры на вкладке с играми пользователя. Игровая сессия симулируется в виде простого диалогового окна, показанного на рисунке 4.12. Все предыдущие сессии можно посмотреть во вкладке *mysessions*. Для каждой сессии отображается название игры, а также время ее начала и конца. Внешний вид списка сессий представлен на рисунке 4.13.

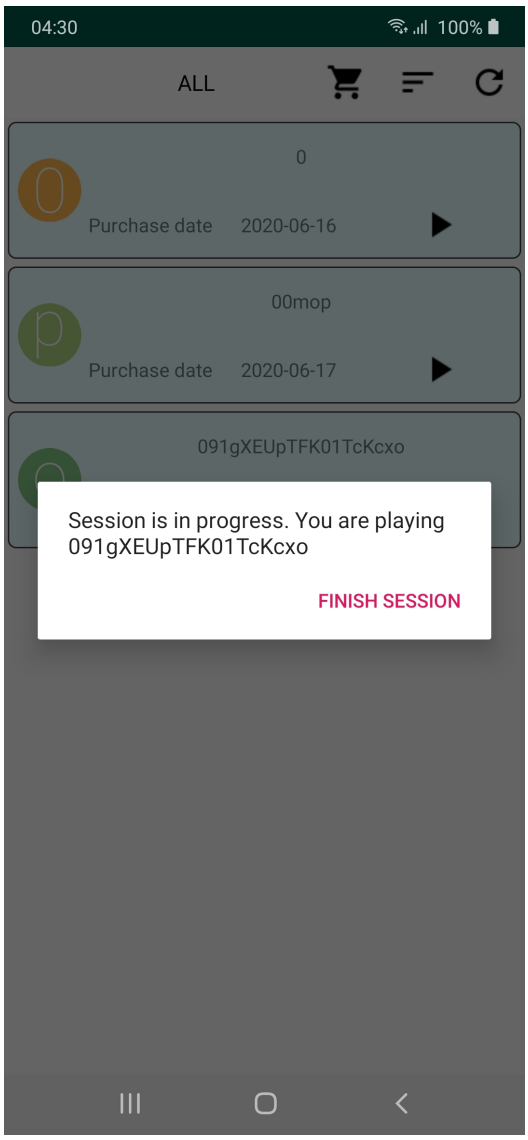

Рисунок 4.12. Симуляция запуска игры

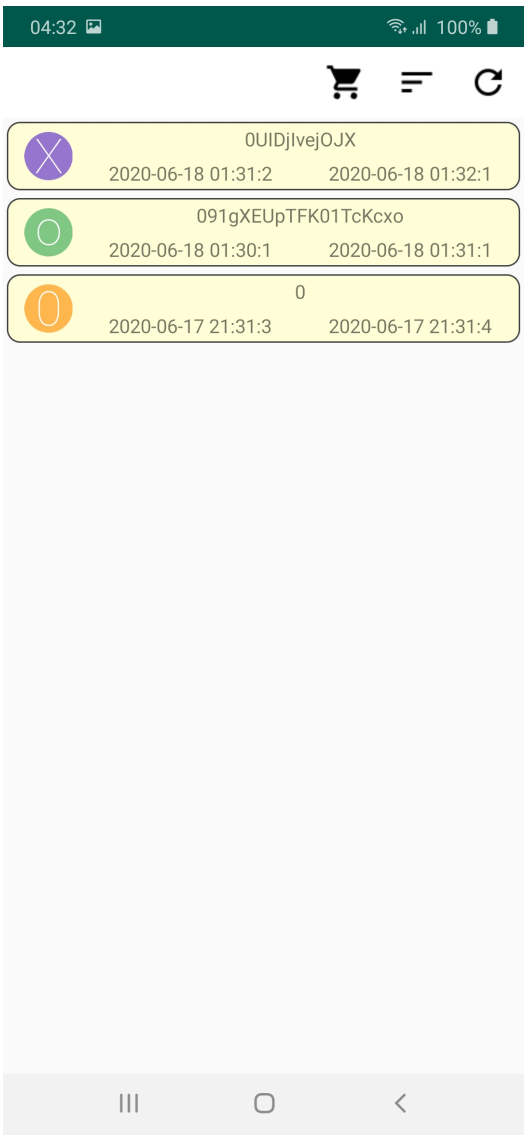

Рисунок 4.13. Список игровых сессий

4.6. Тестирование

Тестирование приложения было проведено вручную. В ходе него были проверены все функции, связанные с взаимодействием с базой данных, а также корректность работы интерфейса.

5. Вывод

В ходе работы были получены навыки по организации взаимодействия Android-приложением с базой данных с использованием http-запросов, и закреплены навыки по работе с базами данных. В ходе разработки приложения были решены различные проблемы, одной из которых является выбор метода авторизации пользователей, в качестве которого было реализовано формирования токена и передачи его клиенту для дальнейшего использования.