Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Компьютерные системы и сети (КСиС)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему:

**КАРТОЧНАЯ ОНЛАЙН-ИГРА**

БГУИР КП 6-05-0612-01-003 ПЗ

Студент Брылёв Н.С.

Руководитель Болтак С.В.

Минск 2025

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 5](#_Toc200231239)

[1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 6](#_Toc200231240)

[1.1 Обзор аналогов 6](#_Toc200231241)

[1.2 Постановка задачи 7](#_Toc200231242)

[2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 9](#_Toc200231243)

[2.1 Структура программы 9](#_Toc200231244)

[2.2 Проектирование интерфейса программного средства 9](#_Toc200231245)

[2.3 Проектирование функционала программного средства 13](#_Toc200231246)

[3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА 13](#_Toc200231247)

[3.3 Работа с базой данных 19](#_Toc200231248)

[3.4 Аутентификация и авторизация 20](#_Toc200231249)

[4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 22](#_Toc200231250)

[5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 24](#_Toc200231251)

[5.1 Интерфейс программного средства 24](#_Toc200231252)

[5.2 Управление программным средством 26](#_Toc200231253)

[ЗАКЛЮЧЕНИЕ 27](#_Toc200231254)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 28](#_Toc200231255)

[ПРИЛОЖЕНИЕ А. Исходный код программы 29](#_Toc200231256)

# ВВЕДЕНИЕ

Современные информационные технологии активно проникают во все сферы жизни человека, предлагая новые возможности не только для работы и повседневных задач, но и для отдыха и развлечений. Одной из популярных форм досуга в цифровую эпоху стали онлайн-игры, позволяющие пользователям со всего мира взаимодействовать и соревноваться в виртуальном пространстве, а также просто приятно проводить время.

С ростом доступности высокоскоростного интернета и развитием веб-технологий возросла потребность в переносе традиционных форм развлечений, таких как настольные и, в частности, карточные игры, в онлайн-формат. Это позволяет не только сохранить популярные игры для новых поколений и сделать их доступными в любое время, но и объединять игроков независимо от их географического положения, создавая новые социальные связи.

В рамках данного курсового проекта рассматривается разработка веб-приложения для популярной карточной игры. Целью проекта является создание интерактивной многопользовательской платформы, где игроки могут создавать игровые комнаты или присоединяться к существующим, и вести игру в соответствии с классическими правилами. Основное внимание уделяется реализации корректной игровой логики, обеспечению взаимодействия игроков в режиме, близком к реальному времени, а также созданию интуитивно понятного пользовательского интерфейса.

Для реализации проекта используются современные технологии и инструменты:

Python — высокоуровневый язык программирования общего назначения, выбранный за его простоту, читаемость и большое количество доступных библиотек.

Django — высокоуровневый веб-фреймворк на Python, позволяющий быстро разрабатывать безопасные и масштабируемые веб-приложения. Он предоставляет мощные инструменты для работы с URL-маршрутизацией, шаблонами, формами и базами данных.

Django ORM — технология, встроенная в Django, позволяющая работать с базой данных (например, информацией об игроках, игровых комнатах, текущем состоянии игр) на уровне объектов Python, абстрагируясь от написания SQL-запросов напрямую.

SQLite — легковесная, файловая система управления базами данных, хорошо подходящая для этапа разработки и для приложений, не требующих высокой параллельной нагрузки на СУБД.

HTML, CSS, JavaScript — стандартные веб-технологии для создания пользовательского интерфейса. JavaScript используется для обеспечения интерактивности на стороне клиента, включая обработку действий игрока и обновление игрового.

# АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## Обзор аналогов

Карточные онлайн-игры представляют собой динамично развивающуюся область игровой индустрии, предлагая игрокам стратегический и тактический геймплей в цифровом формате. В последние годы появилось множество проектов, которые успешно адаптировали классические принципы коллекционных карточных игр в онлайн-среду, обеспечивая удобные механики, глубокий баланс и разнообразные способы взаимодействия между участниками.

Одним из наиболее известных и наиболее схожих приложений является «Durak Online» от R-Soft. Это условно-бесплатная мобильная версия классической карточной игры «Дурак». Игра предлагает различные режимы, включая подкидного и переводного дурака, а также возможность играть с реальными соперниками или против компьютера. Среди плюсов — удобный интерфейс, разнообразие настроек и возможность общения с другими игроками, а среди минусов — наличие внутриигровых покупок, рекламы и ограниченные бесплатные ресурсы.. Внешний вид данного приложения представлен на рисунке 1.1



Рисунок 1.1 – Durak Online

Ещё одним заметным представителем является мобильная карточная игры «UNO!» от Mattel163 Limited. Приложение предлагает классические правила данной карточной игры, а также различные режимы, включая командные матчи и турниры. Среди плюсов — яркий дизайн, удобный интерфейс и возможность играть с друзьями онлайн. Однако минусами могут быть внутриигровые покупки и необходимость подключения к интернету для большинства режимов.



Рисунок 1.2 – UNO!

## Постановка задачи

В рамках данной курсовой работы планируется разработать веб-приложение для многопользовательской карточной онлайн-игры. Приложение должно предоставлять пользователям возможность участвовать в игровых сессиях, взаимодействуя друг с другом в режиме, приближенном к реальному времени. Основной целью является создание интуитивно понятной и функциональной платформы для проведения досуга, с корректной реализацией игровой логики и удобным пользовательским интерфейсом.

Основные функции, подлежащие реализации:

1. **Регистрация и аутентификация пользователей:**
   * Создание учетных записей игроков.
   * Безопасный вход в систему для доступа к игровым функциям.
2. **Управление игровыми комнатами:**
   * Создание новых игровых комнат с указанием параметров (например, количество игроков, сумма ставки в виртуальной валюте).
   * Просмотр списка доступных комнат, ожидающих игроков.
   * Возможность присоединения к существующей комнате.
   * Автоматический или ручной (создателем комнаты) запуск игры при наборе достаточного количества игроков.
3. **Реализация игрового процесса:**
   * Раздача карт игрокам согласно правилам.
   * Определение козырной масти.
   * Реализация ходов игроков (атака, защита, взятие карт, пас) в соответствии с правилами выбранной разновидности карточной игры.
   * Визуализация игрового стола, карт в руке игрока и действий других участников.
   * Автоматическое пополнение руки картами из колоды.
   * Определение победителя и проигравшего в игровой сессии.
4. **Управление профилем игрока:**
   * Отображение основной информации об игроке (например, имя пользователя, баланс виртуальной валюты).

Веб-приложение будет реализовано с использованием следующих технологий:

* Язык программирования: Python.
* Веб-фреймворк: Django, предоставляющий инструменты для маршрутизации, обработки запросов, работы с шаблонами и ORM.
* База данных: SQLite – легковесная, файловая система управления базами данных, используемая для хранения информации об игроках, игровых комнатах, текущем состоянии активных игр и (опционально) истории игр.

# ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

## 2.1 Структура программы

Разрабатываемое веб-приложение построено на основе архитектуры фреймворка Django, который следует принципу MVT (Model-View-Template), являющемуся вариацией MVC (Model-View-Controller). Такая структура обеспечивает четкое разделение логики, данных и представления, что упрощает разработку, поддержку и расширение проекта.

Проект Django

manage.py: Утилита командной строки для взаимодействия с проектом Django

Приложение Сервера

* settings.py: Основной файл конфигурации проекта.
* urls.py (project-level): Главный файл URL-маршрутизации.

Приложение Игры

* models.py Определяет модели данных
* views.py Содержит логику обработки HTTP-запросов
* urls.py (app-level): Файл URL-маршрутизации
* forms.py Определяет формы Django
* game\_logic.py Отдельный модуль, содержащий основную логику самой карточной игры

Приложение Игроков

* models.py

## 2.2 Проектирование интерфейса программного средства

Пользовательский интерфейс веб-приложения для карточной онлайн-игры будет выполнен в простом и интуитивно понятном стиле, ориентированном на удобство игрового процесса и минимизацию отвлекающих элементов. Основные элементы интерфейса будут включать лобби с доступными играми, экран создания комнаты, непосредственно игровой стол и, возможно, страницу профиля игрока.

2.2.1 Лобби (Главная страница после входа)

Лобби является отправной точкой для игрока после успешной аутентификации. Здесь отображается список доступных игровых комнат. Каждая комната в списке будет содержать основную информацию, такую как:

Название комнаты. Количество текущих игроков / максимальное количество игроков. Размер ставки игровой валюты (если есть). Кнопка для присоединения к комнате. Кнопка "Создать комнату": позволяет игроку перейти на страницу создания собственной игровой комнаты. Информация о пользователе: В верхней части страницы (например, в навигационной панели) может отображаться имя текущего пользователя и его баланс виртуальной валюты. Кнопка "Выход": для завершения сессии и выхода из учетной записи. Макет главного окна представлен на рисунке 2.1.

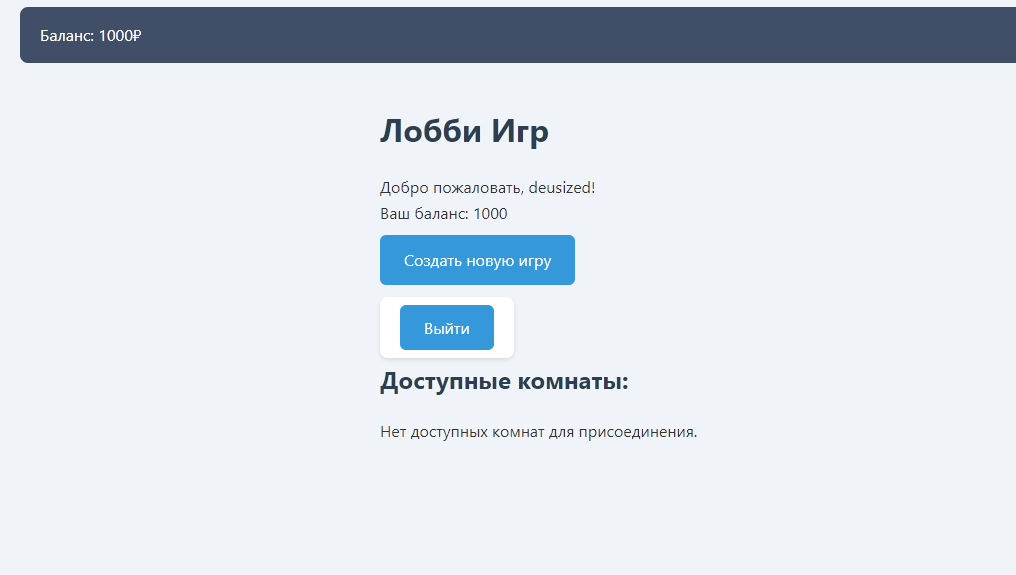


Рисунок 2.1 – Главное окно приложения

2.2.2 Страница создания комнаты

При нажатии на кнопку "Создать комнату" пользователь перенаправляется на страницу, где он может задать параметры новой игры:

Поле для ввода названия комнаты (необязательное). Выбор количества игроков (например, от 2 до 4). Поле для указания размера ставки (виртуальная валюта). Кнопка "Создать" для подтверждения и создания комнаты, после чего игрок автоматически перенаправляется в созданную игровую комнату. Внешний вид данной страницы представлен на рисунке 2.2.

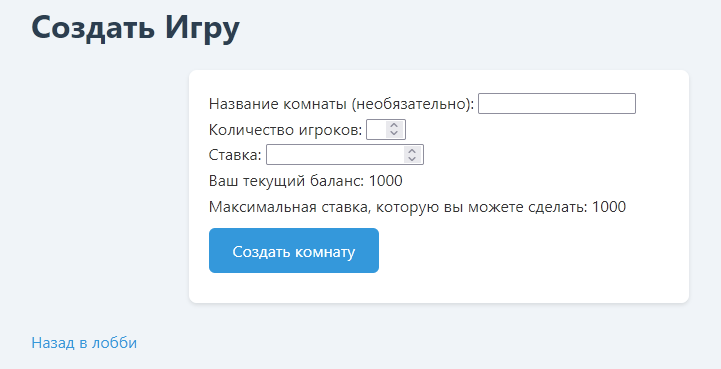


Рисунок 2.2 – Элементы управления приложением

2.2.3 Игровая комната (Игровой стол)

Это основной экран, где происходит игровой процесс. Его интерфейс должен быть максимально наглядным и удобным для быстрых действий. Ключевые элементы:

Игровой стол: Центральная область, где выкладываются атакующие и защищающиеся карты.

Рука текущего игрока: Карты игрока отображаются в нижней части экрана, с возможностью выбора одной или нескольких карт для хода.

Информация о других игроках: Аватары или имена других игроков, количество карт у них на руках (может быть скрыто или показано только число).

Колода и козырная карта: Визуальное представление оставшейся колоды и открытой козырной карты.

Кнопки действий: В зависимости от текущего состояния игры и роли игрока (атакующий, защищающийся), ему доступны кнопки:

"Ходить" / "Подкинуть"

"Отбиться"

"Взять карты"

"Пас" / "Бито"

"Покинуть комнату".

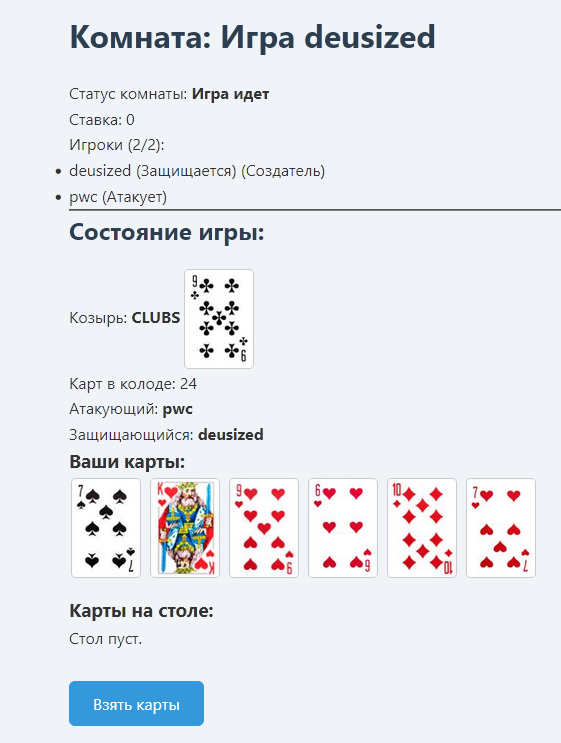
Визуальное оформление интерфейса игрового стола показано на рисунках 2.3-2.4.

Рисунок 2.3 – Интерфейс игрового стола(PC)



Рисунок 2.4 – Интерфейс игрового стола на мобильном устройстве

2.2.4 Окно для регистрации и входа

Окно регистрации пользователя включает в себя поля для ввода логина-никнейма и пароля, а также две кнопки — для регистрации и перехода ко входу. При нажатии на кнопку «Войти» пользователь перенаправляется на окно входа, где предусмотрены поля для ввода имени и почты, а также две кнопки — для входа и регистрации. Внешний вид макетов окон регистрации и входа представлен на рисунке 2.5.

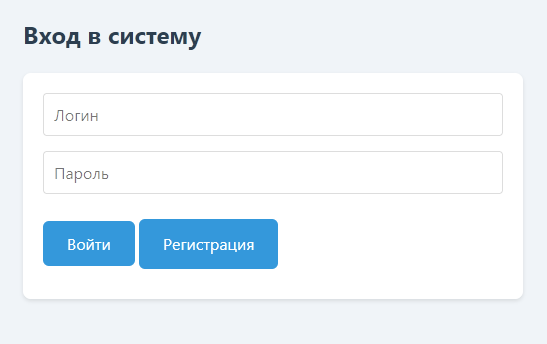
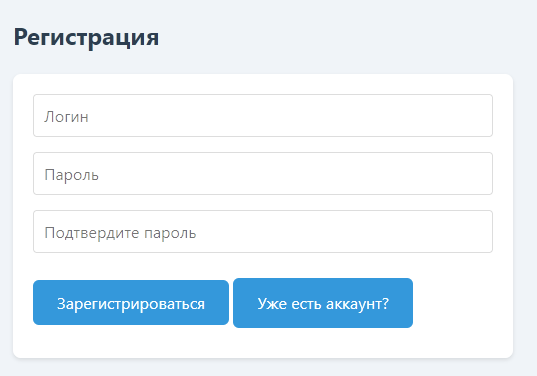


Рисунок 2.5 – Внешний макетов окон регистрации и входа

## 2.3 Проектирование функционала программного средства

Грамотно поставленная задача и четко определенные алгоритмы — ключевые этапы в проектировании программного средства.

Приложение предоставляет пользователю базовый и интуитивно понятный функционал, необходимый для комфортного участия в карточной игре «Дурак» в режиме онлайн с другими игроками. Минимальный набор ключевых функций включает:

* Вход в аккаунт;
* Регистрация нового пользователя
* Просмотр списка игровых комнат;
* Создание новой игровой комнаты;
* Присоединение к существующей игровой комнате;
* Игровой процесс «Дурака»;
* Выход из аккаунта.

2.3.1 Создание новой игровой комнаты

Добавление комнаты будет происходить по нажатию кнопки «Создать игру». После нажатия данной кнопки должно будет открыться окно, в котором пользователь сможет задать название новой комнаты, задать количество игроков, используя ползунок (от 2 до 4 человек), указать сумму ставки, на которую будет проходить игра (от 0 до суммы на счету создающего игру), и добавить комнату в общий список. При добавлении события будет вызываться метод создания комнаты. После выбора настроек пользователь нажимает кнопку «Создать лобби». Блок-схема метода представлена на рисунке 2.6.

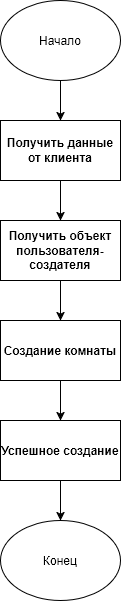


Рисунок 2.6 – Блок-схема метода

# РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

3.1 Выбор технологий и инструментов

Для реализации сайта-игры «Дурак Онлайн» были выбраны современные и широко применяемые технологии, обеспечивающие надёжность, масштабируемость и удобство поддержки проекта. Архитектура приложения построена по клиент-серверной модели. Для взаимодействия в реальном времени между игроками используется технология WebSockets, а для основных операций с данными и управления пользователями – стандартные веб-запросы.

Frontend (пользовательский интерфейс)

Клиентская часть приложения реализована с использованием стандартных веб-технологий:

HTML для структурирования контента страниц игрового интерфейса, лобби и пользовательских форм.

CSS для стилизации и визуального оформления, обеспечивая интуитивно понятный и тематический дизайн.

JavaScript для обеспечения интерактивности пользовательского интерфейса, обработки действий пользователя (выбор карт, нажатие кнопок), отправки данных на сервер через AJAX для таких операций, как создание комнаты или игровые ходы, и для установления и управления WebSocket-соединением для получения обновлений игры в реальном времени.

Данный стек технологий обеспечивает широкую совместимость с современными браузерами, простоту в разработке для поставленных задач и возможность создания динамического интерфейса, необходимого для отображения игрового процесса.

Backend (серверная часть)

В качестве серверной платформы выбран фреймворк Django на языке Python. Этот выбор обусловлен следующими причинами:

Быстрая разработка: Django предоставляет мощные встроенные компоненты, включая ORM для работы с базой данных, систему аутентификации, шаблонизатор и средства для создания форм, что значительно ускоряет разработку.

Надёжность и безопасность: Фреймворк включает защиту от многих распространенных веб-уязвимостей.

Масштабируемость: подходит для дальнейшего развития проекта.

Поддержка WebSockets: Интеграция с Django Channels позволяет реализовать двунаправленную связь в реальном времени между сервером и клиентами, что критично для многопользовательской онлайн-игры.

База данных

Для хранения данных на этапе разработки используется SQLite. Эта файловая реляционная СУБД выбрана за:

Простоту интеграции: не требует отдельной установки и настройки сервера баз данных.

Лёгкость: отлично подходит для разработки и тестирования.

Встроенную поддержку в Python и Django.

В базе данных хранятся:

* Данные пользователей (логин, хэш пароля);
* Данные игровых комнат (создатель, список участников, параметры игры);

Состояние активных игровых партий.

Технологии взаимодействия

HTTP/HTTPS: Основа для большинства взаимодействий – загрузка страниц, аутентификация, создание комнат, а также для отправки игровых действий от клиента на сервер (например, через AJAX-запросы к make\_move\_view).

WebSockets (через Django Channels): используются для мгновенной доставки обновлений состояния игры от сервера всем клиентам в комнате. Это обеспечивает синхронизацию игрового процесса у всех участников без необходимости постоянных запросов со стороны клиента.

JSON: является основным форматом для обмена структурированными данными между клиентом и сервером в AJAX-запросах (например, при совершении хода через make\_move\_view) и в сообщениях, передаваемых по WebSocket.

Такой подход обеспечивает безопасную и масштабируемую работу приложения с учётом аутентификации.

**3.2 Реализация клиентской части**

Клиентская часть сайта-игры «Дурак Онлайн» разработана с использованием стандартных веб-технологий: HTML для разметки структуры страниц, CSS для стилизации и визуального оформления, и нативного JavaScript для обеспечения интерактивности и взаимодействия с сервером. Интерфейс спроектирован таким образом, чтобы быть интуитивно понятным для пользователя и обеспечивать удобный доступ ко всем игровым функциям.

**3.2.1** Структура интерфейса

Пользовательский интерфейс включает следующие основные страницы:

* Страница входа.
* Страница регистрации.
* Главная страница.
* Страница игровой комнаты / Игровой стол.
* Страница создания игровой комнаты.

**3.2.2** Работа с сервером

Взаимодействие клиентской части сайта-игры «Дурак Онлайн» с сервером осуществляется двумя основными способами:

1. Через стандартные HTTP-запросы (GET для получения страниц и данных, POST для отправки форм и выполнения действий). Для некоторых операций, требующих обновления части страницы без полной перезагрузки (например, присоединение к игре, проверка статуса), используются асинхронные запросы (AJAX) с помощью нативного JavaScript.
2. Через WebSocket-соединения для обмена данными в реальном времени во время активной игровой партии.

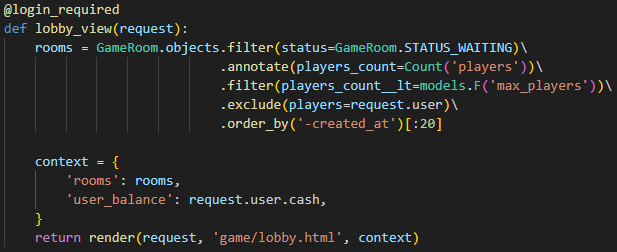


Рисунок 3.1 – Отображение страниц и получение начальных данных

Рисунок 3.2 – Создание новой игровой комнаты



Рисунок 3.3 – Присоединение к игровой комнате

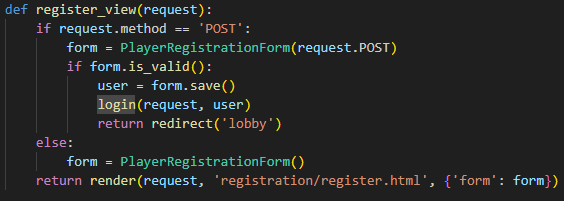


Рисунок 3.4 – Регистрация пользователя

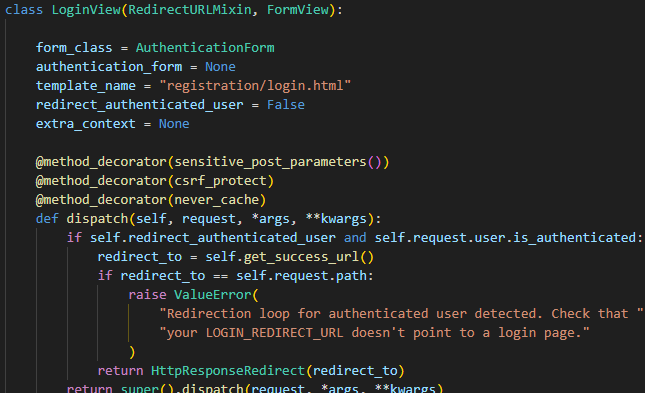


Рисунок 3.5 – Класс входа пользователя

**3.2.3** Обработка токенов и авторизации

Для управления доступом используется встроенная система аутентификации и сессий Django. При успешном входе пользователя сервер создает сессию и устанавливает sessionid cookie в браузере. Этот cookie автоматически передается с последующими HTTP-запросами, позволяя серверу идентифицировать пользователя и проверять его права через request.user. Для WebSocket-соединений AuthMiddlewareStack из Django Channels аналогично предоставляет доступ к аутентифицированному пользователю (scope['user']). Выход из системы завершает сессию пользователя на сервере.

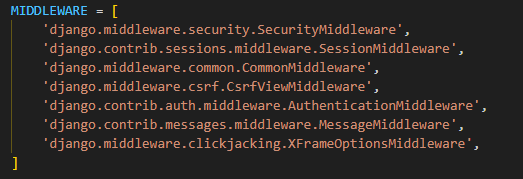


Рисунок 3.7 – Настройки Django middleware

## 3.3 Работа с базой данных

Для хранения и управления данными в сайте-игре «Дурак Онлайн» на этапе разработки используется реляционная база данных SQLite. С её помощью осуществляется долговременное хранение информации об игроках, созданных игровых комнатах, текущем состоянии активных игр и другой необходимой информации.

**3.3.1** Структура базы данных

Структура базы данных определяется моделями Django, которые описаны в файлах models.py соответствующих приложений (players и game). Эти модели отражают сущности предметной области и их взаимосвязи. Основные модели (и соответствующие им таблицы в базе данных) включают:

* + Player
  + GameRoom
  + Game
  + PlayerActivity

**3.3.2** Взаимодействие с базой данных

Взаимодействие серверной части приложения (Django) с базой данных SQLite осуществляется через встроенный Object-Relational Mapper Django.

Модели данных: Классы Python в models.py (описанные выше) определяют структуру таблиц базы данных и связи между ними. Каждая модель соответствует таблице, а атрибуты модели – полям таблицы.

Менеджер моделей и QuerySets: Django предоставляет для каждой модели менеджер (например, Player.objects, GameRoom.objects), через который выполняются запросы к базе данных. Запросы формируются с использованием API QuerySet, который позволяет декларативно описывать операции выборки, фильтрации, сортировки, добавления, обновления и удаления данных. Этот API транслируется Django ORM в соответствующие SQL-запросы.

Миграции: Django имеет встроенную систему миграций. При изменении определения моделей (например, добавлении нового поля или изменении существующего), команда python manage.py makemigrations генерирует файлы миграций, описывающие эти изменения. Команда python manage.py migrate применяет эти миграции к базе данных, обновляя ее схему без потери данных (в большинстве случаев). Это позволяет контролировать версии схемы базы данных и легко применять изменения в различных окружениях.

**3.3.3** Безопасность работы с БД

Для обеспечения безопасности при работе с базой данных реализованы следующие меры:

* Защита от SQL-инъекций: Django ORM по умолчанию использует параметризованные запросы при формировании SQL-команд из QuerySet API.
* Разграничение доступа к данным на уровне бизнес-логики: доступ к игровым комнатам, игровым действиям и данным других пользователей контролируется в представлениях (views.py) и игровой логике (game\_logic.py). Все операции с базой выполняются на стороне сервера, клиент не имеет прямого доступа к БД.

## 3.4 Аутентификация и авторизация

Для обеспечения защищённого доступа к функционалу приложения реализованы механизмы аутентификации (подтверждение личности пользователя) и авторизации (предоставление доступа к определённым данным и действиям в зависимости от прав пользователя).

**3.4.1** Используемые технологии

В проекте для аутентификации и управления сессиями применяется стандартный механизм Django:

Сессии Django: Основной механизм для идентификации пользователя после входа. Сервер создает сессию и связывает ее с пользователем.

Cookie sessionid: Идентификатор сессии хранится в HTTP-only cookie, автоматически отправляемом браузером с каждым запросом. Флаг HttpOnly защищает cookie от доступа через JavaScript, снижая риск XSS-атак.

Middleware Django: SessionMiddleware и AuthenticationMiddleware обрабатывают сессионные cookie для установления request.user.

**3.4.2** Процесс аутентификации

Пользователь вводит свои учётные данные на форме входа. Эти данные отправляются на сервер через HTTP POST-запрос. Сервер Django проверяет подлинность данных: извлекает пользователя по логину и сравнивает предоставленный пароль с хешированным паролем, хранящимся в базе данных.

В случае успеха:

* Django создает или активирует сессию для пользователя.
* Идентификатор сессии (sessionid) отправляется клиенту и сохраняется в браузере в виде cookie.

3.4.3 Управление сессией

Сессионный cookie автоматически прилагается браузером ко всем последующим HTTP-запросам к тому же домену. Django использует этот идентификатор для загрузки данных сессии пользователя на сервере. Срок действия сессии определяется настройками Django. По истечении этого срока или при закрытии браузера (если сессия не является постоянной), сессия становится недействительной, и пользователю потребуется повторно пройти аутентификацию. Явного механизма обновления "токенов", как в JWT, здесь нет; сессия либо активна, либо истекает.

3.4.4 Авторизация

Для защиты API применяется применяется механизм авторизации Django:

* Доступ к защищённым ресурсам разрешён только аутентифицированным пользователям. Это обеспечивается декоратором @login\_required или проверкой request.user.is\_authenticated.
* Сервер, получив sessionid через cookie, определяет текущего пользователя.
* Дальнейшие проверки прав доступа (например, может ли пользователь присоединиться к конкретной комнате или сделать определенный ход) выполняются в логике представлений и игровой логике на основе этого объекта пользователя и правил игры.

3.4.5 Безопасность

Для обеспечения безопасности реализованы следующие меры:

* Хеширование паролей перед хранением в базе данных.
* Сессионные cookie: Идентификатор сессии хранится в cookie, который по умолчанию имеет флаг HttpOnly.
* Защита от подделки сессий: Django использует криптографическую подпись для данных сессии, чтобы предотвратить их подделку.
* Защита от CSRF: Встроенная защита от CSRF-атак для всех POST-запросов.

3.4.6 Logout (выход из системы)

При выходе пользователя:

* Сервер Django завершает текущую сессию пользователя, удаляя связанные с ней данные из серверного хранилища.
* Клиенту отправляется инструкция удалить сессионный cookie (или он становится недействительным).
* Последующие запросы от пользователя будут обрабатываться как от анонимного пользователя, пока он не войдёт в систему снова.

# ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

На этапе тестирования сайта-игры «Дурак Онлайн» проводилась всесторонняя проверка корректности функционирования всех основных компонентов: пользовательского интерфейса (включая отображение карт и комнат), серверной логики (обработка правил игры, создание комнат, управление ставками), системы аутентификации и сессий, а также взаимодействия с базой данных SQLite. Тестирование включало как ручные проверки через веб-интерфейс, так и автоматизированные тесты (если бы они были реализованы) и интерактивное тестирование серверной логики через Django Shell.

Тестирование через Django Shell (python manage.py shell) играло важную роль на ранних этапах для проверки и отладки основного игрового движка (game\_logic.DurakGame) и взаимодействия с моделями данных (Player, GameRoom, Game) до полной готовности пользовательского интерфейса. Это позволяло:

* Программно создавать пользователей и игровые комнаты.
* Имитировать раздачу карт и определение козыря.
* Выполнять игровые ходы (атака, защита, взятие карт, пас), вызывая методы игрового движка напрямую.
* Проверять корректность сохранения и загрузки состояния игры из базы данных.
* Отслеживать изменения баланса игроков и статуса комнат.

В результате комплексного тестирования были выявлены и устранены следующие характерные недочёты:

1. Ошибка: Карты не отображались у второго присоединившегося игрока.
   * Проблема: при тестировании было обнаружено, что если игра начиналась сразу после того, как создатель вошел в комнату, а второй игрок присоединялся позже, то у второго игрока на странице игровой комнаты не отображались его карты, хотя игра уже была в статусе "игра идет".
   * Решение: Логика инициализации DurakGame была перенесена таким образом, чтобы она срабатывала только после фактического начала игры (когда GameRoom.status меняется на PLAYING и все игроки уже находятся в room.players). Конструктор DurakGame теперь корректно получает актуальный список игроков из комнаты и раздает карты именно им при первом создании состояния игры.
2. Ошибка: Некорректное определение первого атакующего игрока.
   * Проблема: в некоторых случаях первый атакующий игрок определялся не по правилу наименьшего козыря, а случайным образом или всегда им становился создатель комнаты, даже если у другого игрока был младший козырь.
   * Решение: Метод \_set\_initial\_attacker\_defender был переписан для более точного поиска наименьшей козырной карты среди всех игроков, находящихся в комнате на момент начала игры. Были добавлены проверки на случай отсутствия козырей у игроков (хотя это маловероятно при стандартной раздаче).
3. Проблема: "Зависание" лобби или некорректное удаление неактивных комнат.
   * Проблема: Комнаты, ожидающие игроков, иногда оставались в списке лобби неопределенно долго, даже если создатель покинул их или все игроки были неактивны.
   * Решение: В модель GameRoom был добавлен метод clean\_up\_inactive\_waiting\_room(), а в представление ping (которое вызывается клиентом для подтверждения активности) добавлена логика вызова этого метода для комнаты, к которой относится пинг. Это позволяет комнатам со статусом "ожидание" автоматически отменяться, если все участники неактивны в течение определенного времени или если комната пуста. Также была улучшена логика в leave\_room, чтобы при выходе создателя ожидающая комната корректно отменялась.
4. Недочет: Отсутствие обратной связи при ошибках на стороне клиента.
   * Проблема: Если пользователь пытался создать комнату со ставкой, превышающей его баланс, или присоединиться к уже заполненной комнате, он не всегда получал понятное сообщение об ошибке, или ошибка отображалась на отдельной странице, а не на той же форме.
   * Решение: В представления create\_room и join\_game была добавлена более детальная проверка условий и использование системы сообщений Django (django.contrib.messages) для вывода уведомлений об ошибках или успехе непосредственно на текущей странице. Соответствующие блоки для отображения сообщений были добавлены в HTML-шаблоны.

Эти и другие, более мелкие, недочеты были устранены в процессе итеративного тестирования, что позволило повысить стабильность и удобство использования сайта-игры.

# РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

## 5.1 Интерфейс программного средства

5.1.1 Окно авторизации и аунтификации

При запуске приложения пользователю сначала отображается окно регистрации. В этом окне необходимо ввести логин и пароль для создания нового аккаунта. После успешной регистрации пользователь автоматически перенаправляется на окно основного лобби. В окне входа осуществляется проверка учетных данных. Если введённые данные корректны, пользователь успешно авторизуется и попадает на окно лобби, где отображается список доступных комнат. Внешний вид окна авторизации веб-браузера представлен на рисунке 5.1.

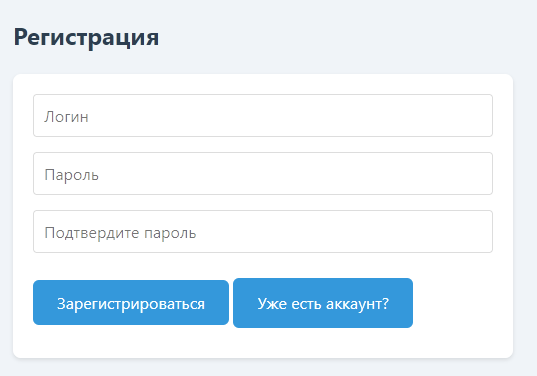


Рисунок 5.1 – Окно регистрации

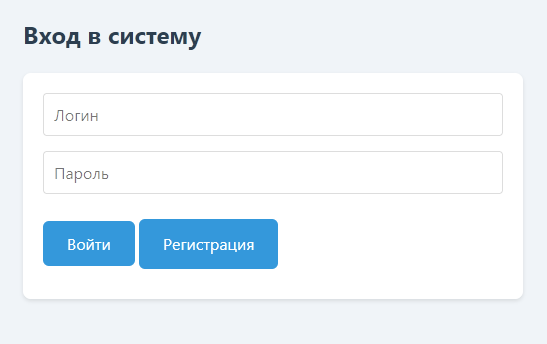


Рисунок 5.2 – Окно входа

5.1.2 Главное окно

Главное окно сайта условно разделено на две главные зоны. В верхней части располагаются основные кнопки “Создать игру” и “Выйти”, а в нижней части — список доступных для присоединения комнат.

Внешний вид главного лобби показан на рисунке 5.3

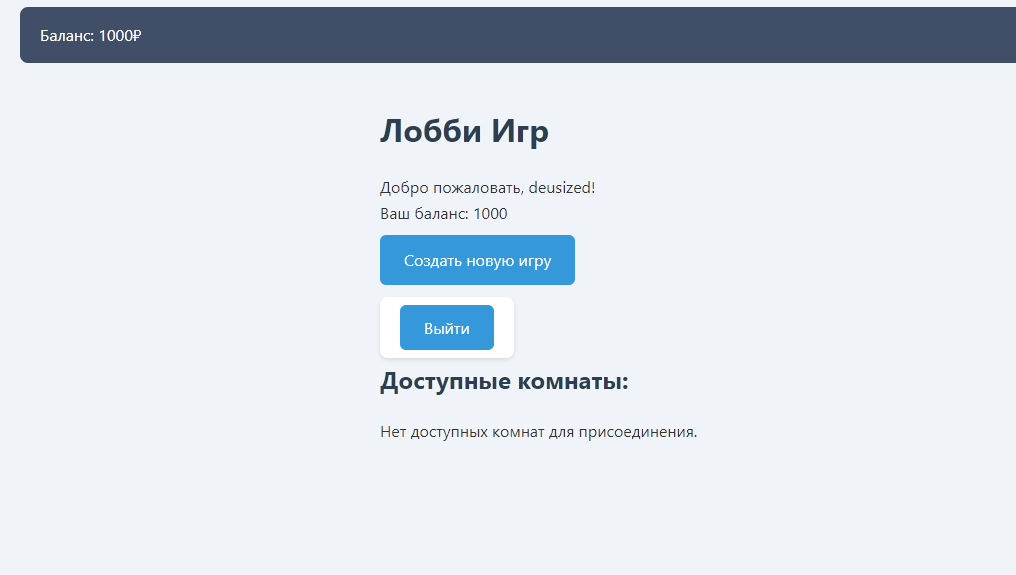


Рисунок 5.3 – Главное лобби

5.1.3 Окно создания новой комнаты

Окно добавления события содержит две кнопки для добавления и отмены создания новой комнаты и поля ввода названия комнаты, количества виртуальной валюты и количества игроков. Внешний вид окна создания новой комнаты представлен на рисунке 5.3.

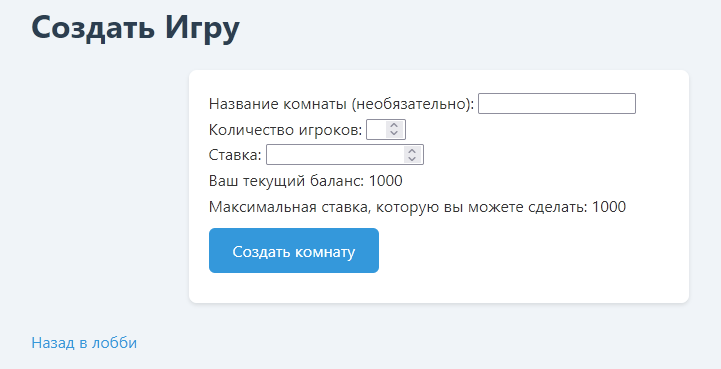


Рисунок 5.3 – Окно создания новой комнаты

5.1.4 Окно игры

Окно игры открывается при создании комнаты или присоединения к ней. Оно используется для основного игрового процесса.

После открывается игровой стол. Внешний вид окна игрового стола представлен на рисунке 5.4.

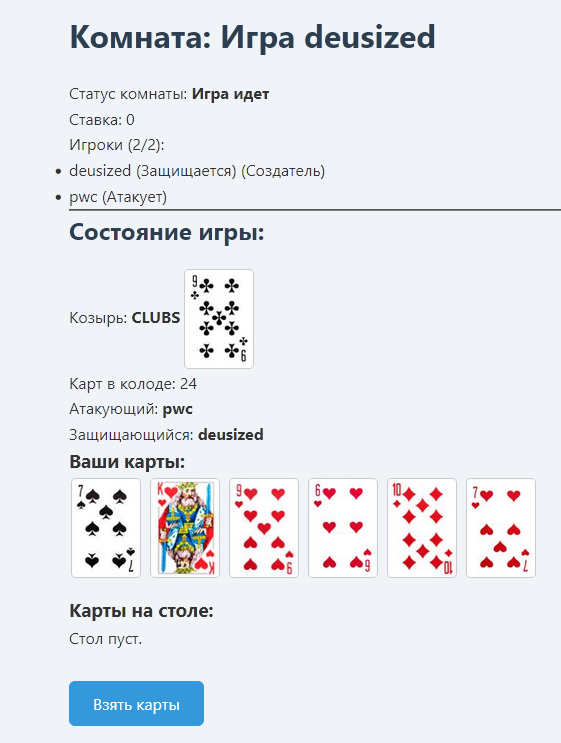


Рисунок 5.4 – Окно игрового стола

## 5.2 Управление программным средством

5.2.1 Элементы управления сайтом-игрой

Интерфейс сайта-игры «Дурак Онлайн» использует стандартные веб-элементы. Ключевые элементы управления включают:

Аутентификация: Формы и кнопки для входа и регистрации. Лобби: Список игр с кнопками «Присоединиться», кнопка «Создать игру» для перехода к форме настройки, кнопка «Выйти». Создание комнаты: Поля для настройки параметров игры (количество игроков, ставка, название) и кнопка «Создать комнату». Игровая комната/Стол: Интерактивные изображения карт в руке игрока, кнопки «Начать игру» (для создателя), «Пас / Бито», «Взять карты», «Покинуть комнату».

Эти элементы обеспечивают навигацию по сайту, управление игровыми сессиями и непосредственное участие в карточной партии.

# ЗАКЛЮЧЕНИЕ

Онлайн-игры, объединяющие пользователей для совместного досуга, продолжают набирать популярность, предоставляя доступные и увлекательные способы взаимодействия. Карточные игры, такие как "Дурак", являются классикой, интерес к которой не ослабевает, а их перенос в цифровой формат открывает новые возможности для игроков.

В рамках данного проекта было разработано программное средство — сайт-игра «Дурак Онлайн», предназначенное для организации и проведения карточных партий между зарегистрированными пользователями в режиме реального времени. Приложение реализовано с использованием стека технологий Python/Django для серверной части, стандартных веб-технологий HTML, CSS и нативного JavaScript для клиентского интерфейса, и SQLite в качестве системы управления базами данных на этапе разработки. Ключевой особенностью является использование Django Channels для обеспечения WebSocket-взаимодействия, необходимого для динамичного игрового процесса.

В процессе разработки были успешно решены основные поставленные задачи. Реализованный функционал сайта-игры включает:

* Регистрацию новых пользователей и аутентификацию существующих.
* Создание игровых комнат с настраиваемыми параметрами (количество игроков, сумма ставки).
* Возможность присоединения к созданным комнатам из общего лобби.
* Полноценный игровой процесс «Дурака» с раздачей карт, определением козыря, ходами атаки и защиты, взятием карт и определением победителя.
* Управление виртуальным игровым балансом пользователей.
* Автоматическое управление жизненным циклом игровых комнат.

Приложение прошло этап начального тестирования, включая проверку игровой логики через Django Shell и тестирование пользовательских сценариев через веб-интерфейс. В ходе этого были выявлены и устранены недочеты, связанные с корректностью раздачи карт при присоединении игроков, определением первого хода и обработкой пользовательских ошибок.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Django Software Foundation. Django - The web framework for perfectionists with deadlines [Электронный ресурс]. – Режим доступа: https://www.djangoproject.com/, свободный.

[2] Python Software Foundation. Python Language Reference [Электронный ресурс]. – Режим доступа: https://docs.python.org/3/reference/index.html, свободный.

[3] Mozilla Developer Network (MDN). HTML: HyperText Markup Language [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/en-US/docs/Web/HTML, свободный.

[4] Mozilla Developer Network (MDN). CSS: Cascading Style Sheets [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/en-US/docs/Web/CSS, свободный.

[5] Mozilla Developer Network (MDN). JavaScript [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/en-US/docs/Web/JavaScript, свободный.

[6] Django Software Foundation. Django Channels - Channels brings event-driven, real-time capabilities to Django [Электронный ресурс]. – Режим доступа: https://channels.readthedocs.io/en/stable/, свободный.

[7] SQLite Consortium. SQLite Home Page [Электронный ресурс]. – Режим доступа: https://www.sqlite.org/index.html, свободный.

# ПРИЛОЖЕНИЕ А. Исходный код программы

Файл game\_logic.py:

from \_\_future\_\_ import annotations

import random

import os

from django.conf import settings

from django.db import transaction

from .models import Game, GameRoom

from players.models import Player

import typing

import logging

logger = logging.getLogger(\_\_name\_\_)

class DurakGame:

def \_\_init\_\_(self, room: GameRoom):

self.room = room

self.game\_model\_instance: typing.Optional[Game] = None

self.players: list[Player] = list(room.players.all().order\_by('id'))

self.player\_hands\_data: dict[str, list[dict]] = {str(p.id): [] for p in self.players}

self.deck: list[dict] = []

self.trump\_suit: typing.Optional[str] = None

self.trump\_card\_revealed: typing.Optional[dict] = None

self.table: list[dict] = []

self.attacker\_index: int = 0

self.defender\_index: int = (self.attacker\_index + 1) % len(self.players) if self.players else 0

self.\_load\_game\_state\_if\_exists()

def \_generate\_deck(self) -> list[dict]:

suits = ['hearts', 'diamonds', 'clubs', 'spades']

ranks = ['6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']

deck = [{'rank': rank, 'suit': suit, 'id': f"{rank}-{suit}"} for suit in suits for rank in ranks]

random.shuffle(deck)

return deck

def \_load\_game\_state\_if\_exists(self):

"""Loads game state from the database if a Game record exists for this room."""

try:

self.game\_model\_instance = Game.objects.get(room=self.room)

self.deck = list(self.game\_model\_instance.deck)

self.trump\_suit = self.game\_model\_instance.trump\_suit

self.table = list(self.game\_model\_instance.table)

self.player\_hands\_data = dict(self.game\_model\_instance.player\_hands)

self.trump\_card\_revealed = self.game\_model\_instance.trump\_card\_revealed

if self.game\_model\_instance.current\_turn:

try:

current\_turn\_user\_id = self.game\_model\_instance.current\_turn.id

self.attacker\_index = next(i for i, p in enumerate(self.players) if p.id == current\_turn\_user\_id)

except (StopIteration, AttributeError):

logger.warning(f"Current turn player {self.game\_model\_instance.current\_turn\_id} not found in room {self.room.id} players. Re-determining attacker.")

self.\_set\_initial\_attacker\_defender()

else:

self.\_set\_initial\_attacker\_defender()

self.defender\_index = (self.attacker\_index + 1) % len(self.players) if self.players else 0

for player\_user in self.players:

if str(player\_user.id) not in self.player\_hands\_data:

self.player\_hands\_data[str(player\_user.id)] = []

logger.info(f"DurakGame state loaded from DB for room {self.room.id}")

except Game.DoesNotExist:

logger.info(f"No existing Game model for room {self.room.id}. DurakGame in pre-init state.")

pass

def initialize\_new\_game\_setup(self):

if self.game\_model\_instance:

logger.warning(f"initialize\_new\_game\_setup called for room {self.room.id}, but Game model already exists. Skipping.")

return

min\_players = getattr(self.room, 'min\_players\_for\_start', 2)

if not self.players or len(self.players) < min\_players:

logger.error(f"Not enough players ({len(self.players)}) to initialize game for room {self.room.id}. Needs {min\_players}.")

return

logger.info(f"Initializing new game setup for room {self.room.id} with {len(self.players)} players.")

self.deck = self.\_generate\_deck()

self.player\_hands\_data = {str(p.id): [] for p in self.players}

self.\_initialize\_hands\_and\_trump()

self.\_set\_initial\_attacker\_defender()

self.game\_model\_instance = Game.objects.create(

room=self.room,

status=GameRoom.STATUS\_PLAYING,

)

self.save\_game\_state()

logger.info(f"New game setup complete and saved for room {self.room.id}. Trump: {self.trump\_suit}. Attacker: {self.players[self.attacker\_index].username if self.players else 'N/A'}")

def \_set\_initial\_attacker\_defender(self):

if not self.players:

self.attacker\_index = 0

self.defender\_index = 0

return

min\_trump\_holder\_idx = -1

min\_trump\_value = 100

hands\_are\_populated = any(self.player\_hands\_data.values())

if self.trump\_suit and hands\_are\_populated:

for idx, player\_user in enumerate(self.players):

player\_hand = self.\_get\_player\_hand(player\_user)

for card in player\_hand:

if card.get('suit') == self.trump\_suit:

card\_val = self.card\_value(card.get('rank', ''))

if card\_val < min\_trump\_value:

min\_trump\_value = card\_val

min\_trump\_holder\_idx = idx

if min\_trump\_holder\_idx != -1:

self.attacker\_index = min\_trump\_holder\_idx

else:

self.attacker\_index = 0

self.defender\_index = (self.attacker\_index + 1) % len(self.players) if self.players else 0

def \_initialize\_hands\_and\_trump(self):

"""Deals initial cards to players and determines the trump card and suit."""

if not self.players or not self.deck:

logger.warning("Cannot initialize hands/trump: no players or empty deck.")

return

for \_ in range(6):

for player\_user in self.players:

if self.deck:

card = self.deck.pop(0)

self.player\_hands\_data.setdefault(str(player\_user.id), []).append(card)

else:

logger.warning("Deck ran out of cards during initial deal.")

break

if not self.deck:

break

if self.deck:

self.trump\_card\_revealed = self.deck[0]

self.trump\_suit = self.trump\_card\_revealed['suit']

elif self.player\_hands\_data and any(self.player\_hands\_data.values()):

logger.warning(f"Deck empty after initial deal for room {self.room.id}. Trump may not be set from deck.")

if not self.trump\_suit:

logger.error(f"CRITICAL: No trump suit could be determined for room {self.room.id}")

else:

self.trump\_suit = None

self.trump\_card\_revealed = None

logger.error(f"Cannot determine trump: deck empty and no cards dealt for room {self.room.id}.")

def \_get\_player\_hand(self, player\_user\_obj: Player) -> list[dict]:

return self.player\_hands\_data.get(str(player\_user\_obj.id), [])

def \_remove\_card\_from\_hand(self, player\_user\_obj: Player, card\_index\_in\_hand: int) -> typing.Optional[dict]:

hand = self.\_get\_player\_hand(player\_user\_obj)

if 0 <= card\_index\_in\_hand < len(hand):

removed\_card = hand.pop(card\_index\_in\_hand)

return removed\_card

logger.warning(f"Invalid card index {card\_index\_in\_hand} for hand of player {player\_user\_obj.id}")

return None

def \_add\_cards\_to\_hand(self, player\_user\_obj: Player, cards\_to\_add: list[dict]):

hand = self.\_get\_player\_hand(player\_user\_obj)

hand.extend(cards\_to\_add)

def card\_value(self, rank\_str: str) -> int:

values = {'6': 6, '7': 7, '8': 8, '9': 9, '10': 10, 'J': 11, 'Q': 12, 'K': 13, 'A': 14}

return values.get(rank\_str.upper(), 0)

def \_can\_beat(self, attack\_card: dict, defense\_card: dict, trump\_suit: typing.Optional[str]) -> bool:

if not trump\_suit:

logger.error("Cannot determine beat: trump\_suit is None.")

return defense\_card['suit'] == attack\_card['suit'] and \

self.card\_value(defense\_card['rank']) > self.card\_value(attack\_card['rank'])

if defense\_card['suit'] == attack\_card['suit']:

return self.card\_value(defense\_card['rank']) > self.card\_value(attack\_card['rank'])

elif defense\_card['suit'] == trump\_suit and attack\_card['suit'] != trump\_suit:

return True

return False

def attack(self, attacking\_player\_user: Player, card\_indices: list[int]) -> dict:

if not self.game\_model\_instance or self.game\_model\_instance.status != GameRoom.STATUS\_PLAYING:

return {'success': False, 'message': "Игра не активна."}

if not self.players or attacking\_player\_user != self.players[self.attacker\_index]:

return {'success': False, 'message': "Сейчас не ваш ход для атаки."}

if not card\_indices:

return {'success': False, 'message': "Нужно выбрать карты для атаки."}

attacker\_hand = list(self.\_get\_player\_hand(attacking\_player\_user))

cards\_to\_play\_objects = []

for idx in sorted(card\_indices, reverse=True):

if 0 <= idx < len(attacker\_hand):

cards\_to\_play\_objects.insert(0, attacker\_hand[idx])

else:

return {'success': False, 'message': f"Неверный индекс карты: {idx}."}

if not cards\_to\_play\_objects:

return {'success': False, 'message': "Не выбрано ни одной валидной карты."}

defender\_user = self.players[self.defender\_index]

defender\_hand\_count = len(self.\_get\_player\_hand(defender\_user))

unbeaten\_attack\_cards\_on\_table = [pair['attack\_card'] for pair in self.table if not pair.get('defense\_card')]

if not self.table or not unbeaten\_attack\_cards\_on\_table:

if not all(c['rank'] == cards\_to\_play\_objects[0]['rank'] for c in cards\_to\_play\_objects):

return {'success': False, 'message': "Для первой атаки все карты должны быть одного ранга."}

if len(cards\_to\_play\_objects) > defender\_hand\_count and defender\_hand\_count > 0 :

return {'success': False, 'message': f"Нельзя атаковать большим количеством карт ({len(cards\_to\_play\_objects)}), чем есть у защищающегося ({defender\_hand\_count})."}

if len(cards\_to\_play\_objects) > 6:

return {'success': False, 'message': "Нельзя атаковать более чем 6 картами за раунд."}

else:

if len(self.table) + len(cards\_to\_play\_objects) > 6:

return {'success': False, 'message': "Слишком много карт на столе (максимум 6)."}

allowed\_ranks\_for\_throw\_in = set()

for pair\_on\_table in self.table:

allowed\_ranks\_for\_throw\_in.add(pair\_on\_table['attack\_card']['rank'])

if pair\_on\_table.get('defense\_card'):

allowed\_ranks\_for\_throw\_in.add(pair\_on\_table['defense\_card']['rank'])

logger.debug(f"Room {self.room.id} - Attempting to throw in. Allowed ranks: {allowed\_ranks\_for\_throw\_in}")

logger.debug(f"Room {self.room.id} - Cards to throw in: {[c['rank'] for c in cards\_to\_play\_objects]}")

if not all(c['rank'] in allowed\_ranks\_for\_throw\_in for c in cards\_to\_play\_objects):

return {'success': False, 'message': "Карты для подкидывания должны совпадать по рангу с картами на столе."}

max\_throw\_in = defender\_hand\_count - len(unbeaten\_attack\_cards\_on\_table)

if len(cards\_to\_play\_objects) > max\_throw\_in and max\_throw\_in >= 0:

return {'success': False, 'message': f"Нельзя подкинуть больше карт ({len(cards\_to\_play\_objects)}), чем может отбить защищающийся ({max\_throw\_in}). Защитнику не хватит карт."}

if max\_throw\_in < 0 and len(cards\_to\_play\_objects) > 0:

return {'success': False, 'message': "Защищающемуся уже не хватает карт отбиться, нельзя подкидывать."}

played\_cards\_count = 0

for card\_idx\_to\_remove in sorted(card\_indices, reverse=True):

removed\_card = self.\_remove\_card\_from\_hand(attacking\_player\_user, card\_idx\_to\_remove)

if removed\_card:

self.table.append({'attack\_card': removed\_card, 'defense\_card': None, 'attacker\_id': attacking\_player\_user.id})

played\_cards\_count += 1

else:

logger.error(f"Failed to remove card at index {card\_idx\_to\_remove} for attack by {attacking\_player\_user.id}")

return {'success': False, 'message': "Внутренняя ошибка: не удалось корректно снять карту с руки."}

if played\_cards\_count == 0 and card\_indices:

return {'success': False, 'message': "Не удалось сыграть выбранные карты."}

self.save\_game\_state()

return {'success': True, 'message': "Атака совершена."}

def defend(self, defending\_player\_user: Player, attack\_card\_table\_index: int, defense\_card\_hand\_index: int) -> dict:

if not self.game\_model\_instance or self.game\_model\_instance.status != GameRoom.STATUS\_PLAYING:

return {'success': False, 'message': "Игра не активна."}

if not self.players or defending\_player\_user != self.players[self.defender\_index]:

return {'success': False, 'message': "Сейчас не ваш ход для защиты."}

if not (0 <= attack\_card\_table\_index < len(self.table)):

return {'success': False, 'message': "Неверный индекс атакующей карты на столе."}

table\_pair = self.table[attack\_card\_table\_index]

if table\_pair.get('defense\_card'):

return {'success': False, 'message': "Эта карта уже отбита."}

attack\_card = table\_pair['attack\_card']

defender\_hand = self.\_get\_player\_hand(defending\_player\_user)

if not (0 <= defense\_card\_hand\_index < len(defender\_hand)):

return {'success': False, 'message': "Неверный индекс карты в руке для защиты."}

defense\_card\_obj = defender\_hand[defense\_card\_hand\_index]

if self.\_can\_beat(attack\_card, defense\_card\_obj, self.trump\_suit):

removed\_defense\_card = self.\_remove\_card\_from\_hand(defending\_player\_user, defense\_card\_hand\_index)

if removed\_defense\_card:

table\_pair['defense\_card'] = removed\_defense\_card

table\_pair['defender\_id'] = defending\_player\_user.id

self.save\_game\_state()

return {'success': True, 'message': "Карта отбита."}

else:

logger.error(f"Internal error removing defense card for player {defending\_player\_user.id}")

return {'success': False, 'message': "Внутренняя ошибка при удалении карты защиты."}

else:

return {'success': False, 'message': "Этой картой нельзя отбиться."}

def \_deal\_cards\_after\_round(self):

if not self.game\_model\_instance or self.game\_model\_instance.status != GameRoom.STATUS\_PLAYING:

return

players\_involved\_in\_round\_needing\_cards: list[Player] = []

attacker\_user = self.players[self.attacker\_index]

if len(self.\_get\_player\_hand(attacker\_user)) < 6:

players\_involved\_in\_round\_needing\_cards.append(attacker\_user)

thrower\_ids\_in\_table = set()

for pair in self.table:

if 'attacker\_id' in pair and pair['attacker\_id'] != attacker\_user.id:

thrower\_ids\_in\_table.add(pair['attacker\_id'])

for p\_user in self.players:

if p\_user.id in thrower\_ids\_in\_table and p\_user not in players\_involved\_in\_round\_needing\_cards:

if len(self.\_get\_player\_hand(p\_user)) < 6:

try:

player\_instance = p\_user

players\_involved\_in\_round\_needing\_cards.append(player\_instance)

except Player.DoesNotExist:

logger.error(f"Player with ID {p\_user.id} not found for dealing cards.")

defender\_user = self.players[self.defender\_index]

if not self.table:

if len(self.\_get\_player\_hand(defender\_user)) < 6 and defender\_user not in players\_involved\_in\_round\_needing\_cards:

players\_involved\_in\_round\_needing\_cards.append(defender\_user)

for player\_to\_deal in players\_involved\_in\_round\_needing\_cards:

hand = self.\_get\_player\_hand(player\_to\_deal)

while len(hand) < 6 and self.deck:

card = self.deck.pop(0)

hand.append(card)

def take\_cards\_action(self, taking\_player\_user: Player) -> dict:

if not self.game\_model\_instance or self.game\_model\_instance.status != GameRoom.STATUS\_PLAYING:

return {'success': False, 'message': "Игра не активна."}

if not self.players or taking\_player\_user != self.players[self.defender\_index]:

return {'success': False, 'message': "Только защищающийся игрок может взять карты."}

if not self.table:

return {'success': False, 'message': "Нет карт на столе, чтобы взять."}

cards\_to\_take\_from\_table = []

for pair in self.table:

cards\_to\_take\_from\_table.append(pair['attack\_card'])

if pair.get('defense\_card'):

cards\_to\_take\_from\_table.append(pair['defense\_card'])

self.\_add\_cards\_to\_hand(taking\_player\_user, cards\_to\_take\_from\_table)

self.table = []

self.\_deal\_cards\_after\_round()

self.attacker\_index = (self.defender\_index + 1) % len(self.players)

self.defender\_index = (self.attacker\_index + 1) % len(self.players)

game\_end\_result = self.\_check\_game\_over\_conditions()

if game\_end\_result and game\_end\_result['game\_over']:

self.save\_game\_state(game\_over\_result=game\_end\_result)

return {\*\*game\_end\_result, 'message': game\_end\_result.get('message', "Игра завершена."), 'success': True}

self.save\_game\_state()

return {'success': True, 'message': "Карты взяты."}

def pass\_or\_bito\_action(self, acting\_player\_user: Player) -> dict:

if not self.game\_model\_instance or self.game\_model\_instance.status != GameRoom.STATUS\_PLAYING:

return {'success': False, 'message': "Игра не активна."}

if not self.table:

return {'success': False, 'message': "Стол пуст, действие 'пас/бито' не применимо в данный момент."}

all\_cards\_on\_table\_defended = all(pair.get('defense\_card') for pair in self.table)

if all\_cards\_on\_table\_defended:

self.table = []

self.\_deal\_cards\_after\_round()

game\_end\_result = self.\_check\_game\_over\_conditions()

if game\_end\_result and game\_end\_result['game\_over']:

self.save\_game\_state(game\_over\_result=game\_end\_result)

return {\*\*game\_end\_result, 'message': game\_end\_result.get('message', "Бито! Игра завершена."), 'success': True}

self.attacker\_index = self.defender\_index

self.defender\_index = (self.attacker\_index + 1) % len(self.players)

self.save\_game\_state()

return {'success': True, 'action\_type': 'bito', 'message': "Бито! Раунд завершен."}

else:

self.save\_game\_state()

return {'success': True, 'action\_type': 'attacker\_passed\_round', 'message': "Атакующий(е) завершили добавление карт. Защищающийся должен отбить оставшиеся или взять."}

def \_check\_game\_over\_conditions(self) -> typing.Optional[dict]:

if not self.game\_model\_instance:

return None

if not self.deck:

players\_with\_cards\_count = 0

last\_player\_with\_cards: typing.Optional[Player] = None

for player\_user in self.players:

if self.\_get\_player\_hand(player\_user):

players\_with\_cards\_count += 1

last\_player\_with\_cards = player\_user

if players\_with\_cards\_count == 0:

return {'game\_over': True, 'is\_draw': True, 'winner': None, 'loser': None, 'message': "Игра окончена! Ничья (все вышли одновременно)."}

if players\_with\_cards\_count == 1:

loser: typing.Optional[Player] = last\_player\_with\_cards

winner: typing.Optional[Player] = None

if len(self.players) == 2:

winner = next((p for p in self.players if p != loser), None)

else:

if self.room.winner and self.room.winner != loser:

winner = self.room.winner

return {'game\_over': True, 'is\_draw': False, 'winner': winner, 'loser': loser,

'message': f"Игра окончена! Проигравший: {loser.username if loser else 'N/A'}."}

if self.deck:

for player\_user in self.players:

if not self.\_get\_player\_hand(player\_user) and not self.room.winner:

pass

return None

def get\_game\_state(self, for\_player\_user\_obj: typing.Optional[Player] = None) -> dict:

"""Возвращает текущее состояние игры, видимое для конкретного игрока."""

game\_status\_from\_model = GameRoom.STATUS\_WAITING

winner\_username = self.room.winner.username if self.room.winner else None

game\_over\_info = None

is\_game\_initialized = bool(self.game\_model\_instance)

if is\_game\_initialized and self.game\_model\_instance:

game\_status\_from\_model = self.game\_model\_instance.status

game\_over\_info = self.\_check\_game\_over\_conditions()

if game\_over\_info and game\_over\_info['game\_over']:

game\_status\_from\_model = GameRoom.STATUS\_FINISHED

winner\_obj\_from\_game\_over = game\_over\_info.get('winner')

if winner\_obj\_from\_game\_over:

winner\_username = winner\_obj\_from\_game\_over.username

elif game\_over\_info.get('is\_draw'):

winner\_username = "Ничья"

else:

game\_status\_from\_model = self.room.status

attacker\_id = self.players[self.attacker\_index].id if self.players and is\_game\_initialized else None

defender\_id = self.players[self.defender\_index].id if self.players and is\_game\_initialized else None

state = {

'room\_id': str(self.room.id),

'players': [],

'attacker\_id': attacker\_id,

'defender\_id': defender\_id,

'attacker\_username': self.players[self.attacker\_index].username if attacker\_id else "N/A",

'defender\_username': self.players[self.defender\_index].username if defender\_id else "N/A",

'trump\_suit': self.trump\_suit,

'trump\_card\_revealed': self.trump\_card\_revealed,

'deck\_count': len(self.deck),

'table': list(self.table),

'status': game\_status\_from\_model,

'winner\_username': winner\_username,

'is\_game\_over': game\_over\_info['game\_over'] if game\_over\_info else False,

'game\_over\_message': game\_over\_info.get('message') if game\_over\_info else None,

'is\_game\_initialized': is\_game\_initialized,

}

for idx, p\_user\_loop in enumerate(self.players):

hand\_cards = self.\_get\_player\_hand(p\_user\_loop)

player\_data = {

'id': p\_user\_loop.id,

'username': p\_user\_loop.username,

'card\_count': len(hand\_cards),

'is\_current\_player\_for\_state': p\_user\_loop == for\_player\_user\_obj,

'cards': []

}

if is\_game\_initialized and (p\_user\_loop == for\_player\_user\_obj or game\_status\_from\_model == GameRoom.STATUS\_FINISHED):

player\_data['cards'] = []

for card\_idx\_in\_hand, card\_in\_hand in enumerate(hand\_cards):

card\_data\_to\_append = {

'rank': card\_in\_hand['rank'],

'suit': card\_in\_hand['suit'],

'id': card\_in\_hand.get('id', f"{card\_in\_hand['rank']}-{card\_in\_hand['suit']}"),

'image\_url': self.\_get\_card\_image\_url(card\_in\_hand),

'hand\_index': card\_idx\_in\_hand

}

player\_data['cards'].append(card\_data\_to\_append)

state['players'].append(player\_data)

if is\_game\_initialized:

for table\_pair\_in\_state in state['table']:

if table\_pair\_in\_state.get('attack\_card'):

table\_pair\_in\_state['attack\_card']['image\_url'] = self.\_get\_card\_image\_url(table\_pair\_in\_state['attack\_card'])

if table\_pair\_in\_state.get('defense\_card'):

table\_pair\_in\_state['defense\_card']['image\_url'] = self.\_get\_card\_image\_url(table\_pair\_in\_state['defense\_card'])

if state['trump\_card\_revealed'] and isinstance(state['trump\_card\_revealed'], dict):

state['trump\_card\_revealed']['image\_url'] = self.\_get\_card\_image\_url(state['trump\_card\_revealed'])

return state

def \_get\_card\_image\_url(self, card\_dict: dict) -> str:

if not card\_dict or not card\_dict.get('suit') or not card\_dict.get('rank'):

return os.path.join(settings.STATIC\_URL, 'cards/back.png')

suit = card\_dict['suit'].lower()

rank = card\_dict['rank'].upper()

return f"{settings.STATIC\_URL}cards/{suit}/{rank}.png"

def save\_game\_state(self, game\_over\_result: typing.Optional[dict] = None):

if not self.game\_model\_instance:

logger.warning(f"Attempted to save game state for room {self.room.id}, but no Game model instance exists.")

return

with transaction.atomic():

game = self.game\_model\_instance

current\_attacker\_user: typing.Optional[Player] = self.players[self.attacker\_index] if self.players and 0 <= self.attacker\_index < len(self.players) else None

game.current\_turn = current\_attacker\_user

game.trump\_suit = self.trump\_suit

game.trump\_card\_revealed = self.trump\_card\_revealed

game.deck = self.deck

game.table = self.table

game.player\_hands = self.player\_hands\_data

is\_game\_truly\_over = game\_over\_result and game\_over\_result.get('game\_over', False)

if is\_game\_truly\_over:

game.status = GameRoom.STATUS\_FINISHED

self.room.status = GameRoom.STATUS\_FINISHED

winner\_obj: typing.Optional[Player] = game\_over\_result.get('winner')

loser\_obj: typing.Optional[Player] = game\_over\_result.get('loser')

is\_draw = game\_over\_result.get('is\_draw', False)

if hasattr(self.room, 'end\_game\_from\_logic'):

self.room.end\_game\_from\_logic(winner=winner\_obj, loser=loser\_obj, is\_draw=is\_draw, final\_pot\_value=None)

elif hasattr(self.room, 'end\_game'):

self.room.end\_game(winner=winner\_obj)

else:

if winner\_obj and not self.room.winner:

self.room.winner = winner\_obj

self.room.save(update\_fields=['status', 'winner'] if winner\_obj and not is\_draw else ['status'])

else:

game.status = GameRoom.STATUS\_PLAYING

self.room.status = GameRoom.STATUS\_PLAYING

if self.deck:

for p\_user in self.players: # p\_user is Player

if not self.\_get\_player\_hand(p\_user) and not self.room.winner:

self.room.winner = p\_user

logger.info(f"Player {p\_user.username} is out of cards (deck not empty), marked as potential winner for room {self.room.id}.")

break

self.room.save(update\_fields=['status', 'winner'] if self.room.winner else ['status'])

game.save()