МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения (ИиППО)

КУРСОВАЯ РАБОТА

по дисциплине: Разработка клиент-серверных приложений
по профилю: Разработка программных продуктов и проектирование информационных систем
направления профессиональной подготовки: 09.03.04 «Программная инженерия»

Тема: «Разработка клиент-серверного приложения игра кликер»

Студент: Толмач Владимир Викторович
Группа: ИКБО-36-22
Работа представлена к защите 20.05.2025 /_____/Толмач В.В./
(подпись и ф.и.о. студента)

Руководитель: старший преподаватель Синицын Анатолий Васильевич

Работа допущена к защите__.06.2025 /_____ /Синицын А.В./
(подпись и ф.и.о. рук-ля)

Оценка по итогам защиты: _____
_____ /___.06.2025,_____/
_____ /___.06.2025, Синицын А.В., старший преподаватель/
(подписи, дата, ф.и.о., должность, звание, уч. степень двух преподавателей, принявших защиту)

М. РТУ МИРЭА. 2025

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА - Российский технологический университет»
РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения (ИиППО)

ЗАДАНИЕ
на выполнение курсовой работы

по дисциплине: Разработка клиент-серверных приложений
по профилю: Разработка программных продуктов и проектирование информационных систем
направления профессиональной подготовки: Программная инженерия (09.03.04)
Студент: Толмач Владимир Викторович
Группа: ИКБО-36-22
Срок представления к защите: 20.05.2025
Руководитель: старший преподаватель Синицын А.В.

Тема: «Разработка клиент-серверного приложения игра кликер».

Исходные данные: Go, Git, PostgreSQL, NextJS, нормативный документ: инструкция по организации и проведению курсового проектирования СМКО МИРЭА 7.5.1/04.И.05-18.

Перечень вопросов, подлежащих разработке, и обязательного графического материала: 1. Провести анализ предметной области для выбранной темы. 2. Выбрать клиент-серверную архитектуру для разрабатываемого приложения и дать её детальное описание с помощью UML. 3. Выбрать программный стек для реализации фуллстек CRUD приложения. 4. Разработать клиентскую и серверную части приложения, реализовать авторизацию и аутентификацию пользователя, обеспечить работу с базой данных, заполнить тестовыми данными, валидировать ролевую модель на некорректные данные. 5. Провести фаззинг-тестирование. 6. Разместить исходный код клиент-серверного приложения в репозитории GitHub с наличием Dockerfile и описанием структуры проекта в readme файле. 7. Развернуть клиент-серверное приложение в облаке. 8. Разработать презентацию с графическими материалами.

Руководителем произведён инструктаж по технике безопасности, противопожарной технике и правилам внутреннего распорядка.

Зав. кафедрой ИиППО: _____ / *Болбаков Р. Г.* /, «21» февраля 2025 г.
Задание на КР выдал: _____ / *Синицын А. В.* /, «21» февраля 2025 г.
Задание на КР получил: _____ / *Толмач В. В.* /, «21» февраля 2025 г.

АННОТАЦИЯ

Отчет 334с., 24 рис., 5 табл., 12 источн.

ВЕБ-ПРИЛОЖЕНИЕ, TELEGRAM, CLICKER, GO, NEXT, ИГРА, КЛИКЕР

Объект исследования – клиент-серверное приложение игра кликер.

Цель работы – анализ, проектирование и реализация серверной части и клиентской части приложения для игры.

В ходе работы был проведен анализ предметной области, сформированы требования продукту, выбрана методология реализации, создана архитектура, реализована серверная часть и клиентская часть интернет-ресурса на основе созданной архитектуры.

С учетом популярности подобных игр, в перспективе ожидается постоянный рост числа пользователей веб-приложения.

ANNOTATION

Report 34 p., 24 figures, 5 tables, 12 sources.

WEB APPLICATION, TELEGRAM, CLICKER, GO, NEXT, GAME, CLICKER

The object of the study is a client–server application game clicker.

The purpose of the work is to analyze, design and implement the server side and the client side of the game application.

In the course of the work, an analysis of the subject area was carried out, product requirements were formed, an implementation methodology was selected, an architecture was created, the server part and the client part of the Internet resource were implemented based on the created architecture.

Given the popularity of such games, a steady increase in the number of users of the web application is expected in the future.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящем отчете применяются следующие термины с соответствующими определениями:

Клиент	– программное обеспечение, запрашивающее ресурсы или услуги у сервера в сети
Паттерн	– повторяемое решение типичной задачи проектирования в программировании
Use-Case	– описание того, как пользователь взаимодействует с системой, чтобы достичь цели

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящем отчете применяются следующие сокращения и обозначения:

СУБД	–	система управления базами данных
API	–	Application Programming Interface
MVC	–	Model-View-Controller
SQL	–	Structured Query Language
VPS	–	Virtual Private Server

ВВЕДЕНИЕ

Целью данной курсовой работы является анализ, проектирование и реализация клиентской и серверной части приложения «Игра кликер». Проект предполагает проектирование архитектуры и функциональности приложения с использованием языка программирования Go [12], фреймворка NextJS [1] и базы данных PostgreSQL [3]. Для тестирования будет использоваться ручное тестирование, для контейнеризации и развертывания Docker [4].

Для достижения данной цели были поставлены следующие задачи:

1) проанализировать предметную область разрабатываемого приложения, выделить функциональные и нефункциональные требования, ограничения к системе,

2) оценить архитектурные паттерны, разработать архитектуру приложения,

3) проанализировать доступные технологии для реализации приложения на выбранной архитектуре, разработать приложение,

4) провести тестирование приложения.

Для достижения поставленных задач используются методы анализа и сравнительного исследования. Результаты данной работы могут быть полезными для разработчиков приложений.

В разделе с описанием предметной области расположена информация об анализе предметной области, функциональные и нефункциональные требования к системе, а также ограничения системы.

В разделе разработки архитектуры указаны сведения об оценке архитектурных стилей и разработке архитектуры, на основе выбранного стиля.

В разделе разработки представлен процесс разработки и тестирования.

1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Анализ предметной области

Анализ предметной области проводился среди приложений на тематику «Игра кликер». Были выбраны интернет-ресурсы: Notcoin [5], HamsterKobmat [6] и Cookie clicker [7], представленные на рисунках 1.1-1.3.

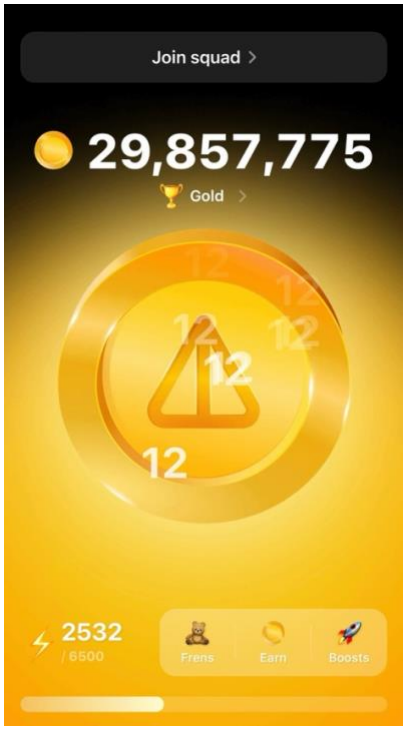


Рисунок 1.1 – Клиентская часть сайта Notcoin

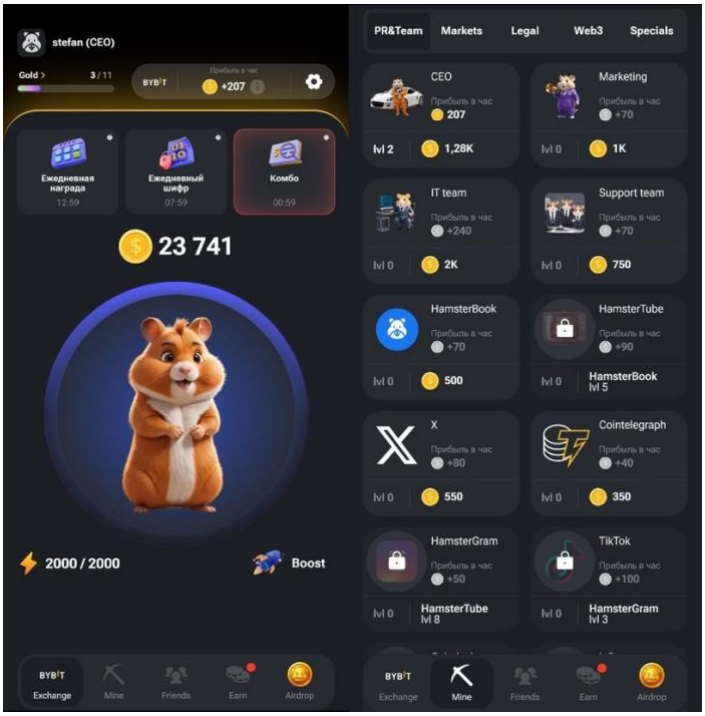


Рисунок 1.2 – Клиентская часть сайта HamsterKobmat

Рисунок 1.3 – Клиентская часть сайта Cookie clicker

Все перечисленные веб-приложение имеют регистрацию, вход в аккаунт, возможность игры, покупок и доступа к личной статистике.

Исходя из анализа данных веб-приложений следует отметить, что разрабатываемое веб-приложение данной тематики должно обладать следующими возможностями:

– регистрация нового аккаунта,

– вход в существующий аккаунт,

– получение информации о пользователе,

– возможность покупки улучшений,

– просмотр и анализ топа игроков.

Также приложение должно иметь способность конкурировать на современном рынке веб-приложений данной тематики. В связи с этим, при разработке учитываются все современные возможности: удобный интерфейс, отзывчивый дизайн, сбор и отображение полной информации о действиях и

прогрессе игрока. Уникальность решения заключается в улучшенной внутриигровой экономике, способствующей более сбалансированному и увлекательному игровому процессу, а также в системе активного топа игроков, которая повышает соревновательность и вовлечённость пользователей.

Кроме того, особое внимание уделяется технической составляющей системы, так как приложение проектируется таким образом, чтобы обеспечить возможность дальнейшего расширения функционала и минимизации затраченного на это времени.

1.2 Функциональные требования к системе

Система игры-кликера предоставляет пользователям возможность регистрации и управления своим игровым профилем. Это включает создание уникального игрового аккаунта, в котором хранятся ключевые параметры игрока: прогресс, внутриигровые ресурсы, достижения и статистика. Пользователи смогут активно взаимодействовать с интерфейсом игры: накапливать ресурсы через клики, улучшать параметры, участвовать в соревновательном рейтинге.

1.3 Нефункциональные требования к системе

Система должна обеспечивать высокую скорость работы, обрабатывая запросы с минимальными задержками и предоставляя доступ к данным в режиме реального времени.

Система должна поддерживать увеличение числа пользователей, что позволит добавлять новые функции и расширять её возможности без снижения производительности.

Регулярное тестирование и своевременное выявление потенциальных проблем обеспечат высокую надёжность и минимизируют время простоя.

1.4 Ограничения системы

Ограничения системы касаются поддерживаемых форматов и единиц измерения.

С точки зрения технических ограничений, система будет функционировать в условиях ограниченных серверных ресурсов, таких как оперативная память, минимум 4 Gb, и процессорное время, 500 секунд CPU в сутки. Это потребует оптимизации алгоритмов для минимизации нагрузки и поддержания стабильности.

Также необходимо определить сроки выполнения проекта. Так, основные этапы разработки и тестирования, должны быть выполнены не более чем за 2-3 месяца.

2 РАЗРАБОТКА АРХИТЕКТУРЫ

2.1 Оценка архитектурных паттернов

Существует множество архитектурных паттернов, каждый из которых имеет свои особенности, преимущества и недостатки.

Одним из самых распространенных является MVC [8]. Этот паттерн разработки нужен для того, чтобы разделить логические части приложения и создавать их отдельно друг от друга. То есть писать независимые блоки кода, которые можно менять, как угодно, не затрагивая другие.

Следующим архитектурным паттерном является клиент-серверная архитектура [9]. В данном стиле приложение разделяется на две основные части, а именно клиент, который запрашивает услуги, и сервер, который их предоставляет. Преимущества клиент-серверного паттерна заключаются в четком разделении ролей между клиентом и сервером, что упрощает поддержку и развитие.

Многоуровневая архитектура [10] представляет собой паттерн, в котором разделяются функции представления, обработки и хранения данных. Такой архитектурный паттерн смягчает возрастающую сложность приложений, упрощает и делает гибче их доработку. Такая архитектура состоит из различных уровней, каждый из которых соответствует отдельной функции. Таким образом, вносить изменения в каждый отдельный уровень проще, чем заниматься всей архитектурой. Разработчики могут создавать гибкие и повторно используемые приложения. Это значительно упрощает весь процесс управления системой.

Микросервисная архитектура [11] предполагает, что приложение делится на множество небольших, самостоятельных сервисов, каждый из которых отвечает за свою сферу деятельности. Микросервисы обеспечивают высокую гибкость и возможность использования различных технологий и языков программирования для каждого сервиса. Однако такое разнообразие сервисов может усложнить управление системой и вызвать трудности с тестированием и отладкой.

В таблице 1 приведены характеристики по критериям к каждому архитектурному паттерну.

Таблица 1 – Характеристики архитектурных паттернов по критериям

Критерии	MVC	Клиент-серверная	Многоуровневая	Микросервисная
Масштабируемость	Хорошо масштабируется по горизонтали, но может быть сложно при росте кодовой базы	Ограничена мощностью сервера	Лучше масштабируется за счет четкого разделения слоев	Независимое масштабирование отдельных микросервисов
Производительность	Быстрее в небольших проектах, но может страдать от накладных расходов при сложной логике	Зависит от сети и сервера, возможны задержки при большом количестве клиентов	Может быть медленнее из-за межслойного взаимодействия, оптимизируется за счет распределения нагрузки	Может снижаться из-за сетевых вызовов, но компенсируется независимой оптимизацией сервисов
Простота разработки и поддержки	Проще для небольших проектов, но при росте может стать сложным	Разделение ролей упрощает поддержку и развитие	Четкое разделение обязанностей упрощает поддержку, но требует больше начальных усилий	Отдельные сервисы проще развивать и поддерживать
Гибкость	Гибкий в рамках одного слоя, но сложнее менять архитектуру	Ограничения по технологиям сервера и клиента	Более гибкая, так как слои можно заменять независимо	Возможность использования разных технологий для разных задач

В контексте разработки серверной части приложения для мониторинга здоровья, выбор клиент-серверной архитектуры обоснован несколькими ключевыми факторами в ходе сравнительного анализа. В сфере здравоохранения необходимо быстро адаптироваться к изменениям в бизнес-требованиях и объемах данных, чего позволяет добиться клиент-серверная архитектура.

Клиент-серверная архитектура позволяет минимизировать время добавления нового функционала, что может улучшить поддержку приложения и стоимость его разработки.

Также клиент-серверная архитектура полностью удовлетворяет функциональным, нефункциональным требованиям системы, так как данная архитектура сможет обеспечить высокую скорость работы, обрабатывая запросы с минимальными задержками, может легко адаптироваться к увеличению количества пользователей и объёма данных.

2.2 Основные компоненты приложения

Основные компоненты приложения включают в себя следующие элементы:

1) клиент, предоставляющий доступ ко всем возможностям приложения через взаимодействие с серверной частью, используя разработанное API;

2) сервер, выполняет роль логической части приложения, которая отвечает за непосредственную реализацию управления доступом к данным приложения;

3) база данных, для хранения всей информации приложения.

2.3 Сценарии взаимодействия с приложением

Для пользования разрабатываемым приложением, пользователь может различными способами взаимодействовать с приложением. На рисунке 2.1 изображена диаграмма Use-Case, на которой отображены основные сценарии взаимодействия с системой.

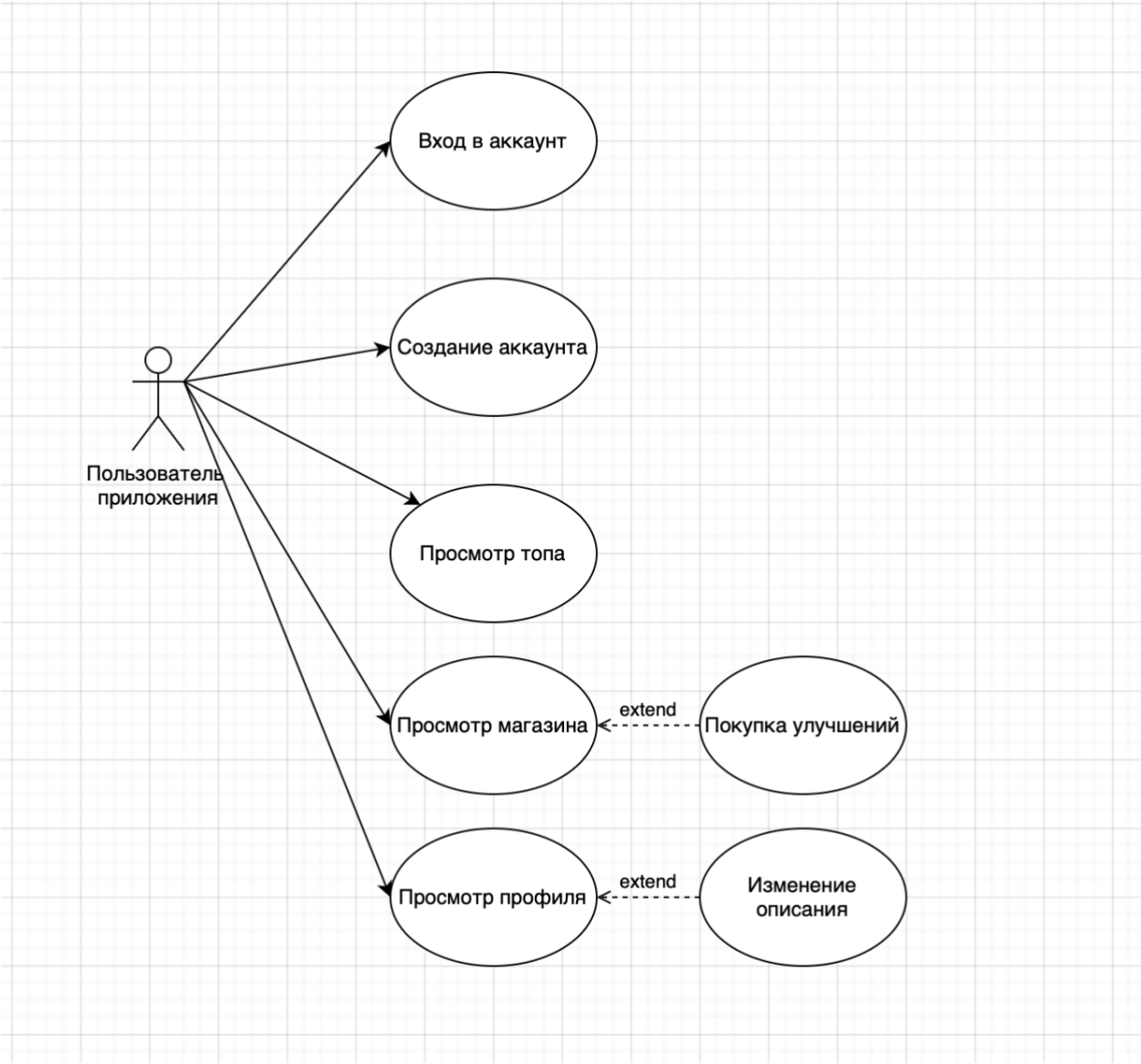


Рисунок 2.1 – Use-Case диаграмма системы

2.4 Модель базы данных

В системе реализована работа с базой данных с помощью СУБД PostgreSQL [3]. На рисунке 2.2 представлена таблица для хранения персональных данных, таких как имя (username), Телеграм ID, описания и баланса.

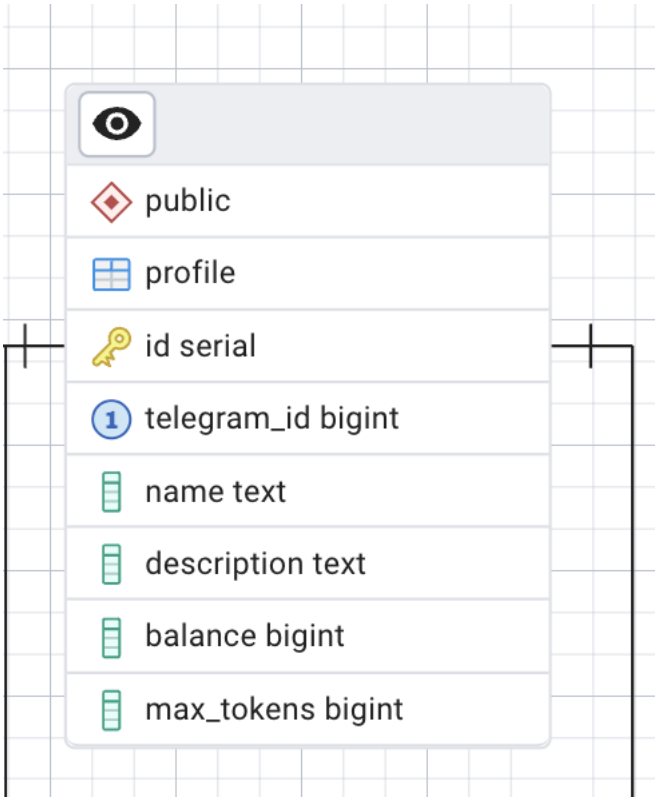


Рисунок 2.2 – Модель таблицы персональных данных

На рисунке 2.3 представлена таблица для хранения покупок и предметов в магазине.

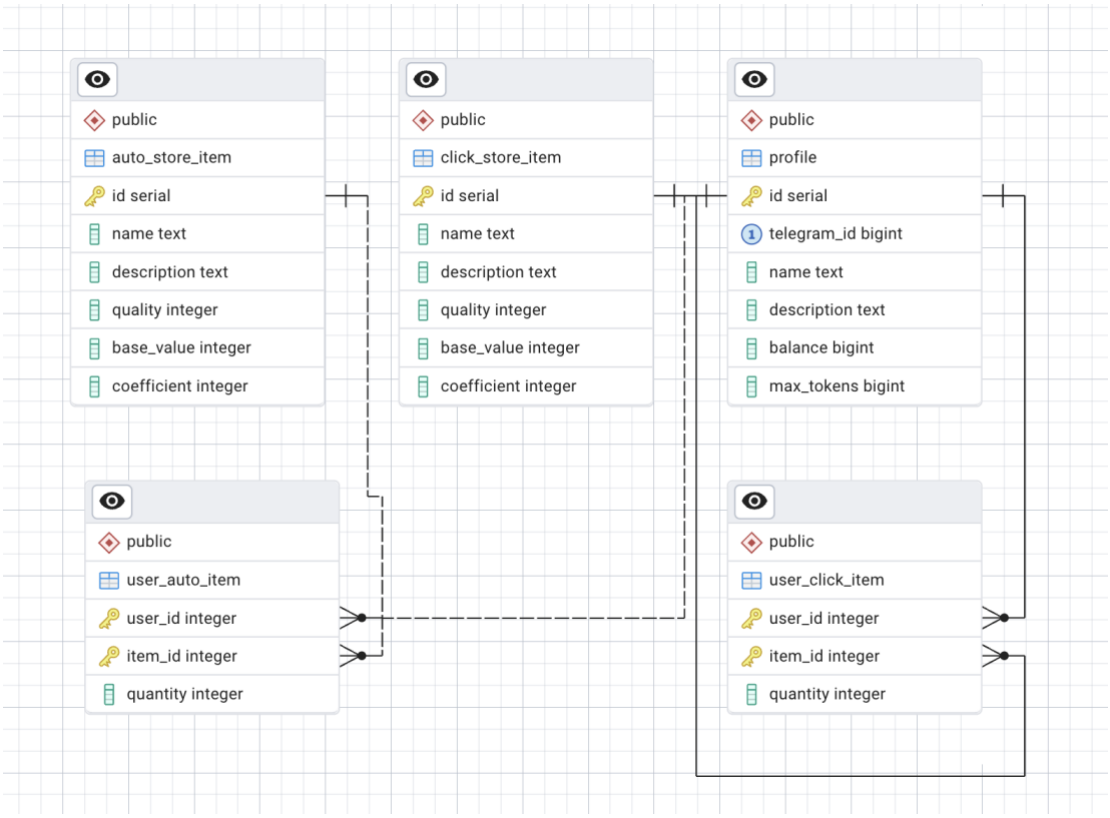


Рисунок 2.3 – Модели таблицы работы с магазином

2.5 Физическая инфраструктура

Для разработки и развертывания приложения можно использовать различные типы физической инфраструктуры, такие как серверы под управлением операционных систем Linux или Windows, VPS или традиционные веб-хостинги.

В рамках разработки данной системы планируется использовать удаленный компьютер в качестве сервера, на котором будут функционировать все сервисы приложения и базы данных. Локальный сервер удовлетворяет техническим ограничениям системы, так как имеет 32 Gb оперативной памяти и CPU с 8 ядрами.

3 ОБОСНОВАНИЕ ВЫБОРА ТЕХНОЛОГИЙ

3.1 Серверная часть

При выборе технологий для реализации серверной части приложения, стоит рассмотреть несколько популярных языков программирования и их фреймворки, так как каждый язык обладает собственными преимуществами и недостатками для создания сложных приложений.

Один из широко используемых языков для серверной части – Java [12]. Его ключевыми преимуществами являются зрелость экосистемы, обширная база библиотек и фреймворков, а также активное сообщество разработчиков.

Python [1] также является популярным выбором для создания серверной части любых приложений. Его простота и высокая скорость разработки привлекают разработчиков.

Golang (Go) [13] стал одним из предпочтительных языков для серверной части, благодаря своей производительности, встроенной поддержке параллелизма и низким требованиям к ресурсам.

В таблице 3 представлена сравнительная характеристика описанных языков программирования в рамках серверной части.

Таблица 3 – Сравнительная характеристика языков программирования

Язык	Преимущества	Недостатки	Примеры фреймворков
Java	Зрелость экосистемы, обширная база библиотек и фреймворков	Избыточно ресурсоемкий, высокие требования к памяти и производительности	Spring Boot
Python	Простота и скорость разработки	Уступает в производительности, ограниченные возможности в условиях высокой нагрузки	Django, FastAPI
Golang (Go)	Высокая производительность, поддержка параллелизма, низкие требования к ресурсам	Менее богатая экосистема по сравнению с другими языками, ограниченная нативная поддержка фреймворков	Gin, Echo

Сравнивая Go с другими языками, можно выделить высокую производительность и надежность, простоту в разработке. Это особенно важно для серверной части приложений.

3.2 Клиентская часть

Клиентская часть приложения реализована с использованием Next.js — современного фреймворка на базе React, который обеспечивает высокую производительность, удобство маршрутизации и серверный рендеринг.

В таблице 4 представлена сравнительная характеристика популярных клиентских технологий.

Таблица 4 – Сравнительная характеристика клиентских технологий

Название СУБД	Преимущества	Недостатки
Next.js	SSR, SSG, отличная производительность, хорошая SEO-оптимизация, удобная маршрутизация	Более сложная настройка по сравнению с чистым React
Vue.js	Легко осваивается, реактивность, простота структуры	Меньшее сообщество, меньшее количество крупных проектов
Angular	Встроенные решения "из коробки", хорошая структура проекта	Высокий порог входа, громоздкость

Пользовательский интерфейс позволяет игрокам регистрироваться, выполнять игровые действия, просматривать статистику, рейтинг, а также получать рекомендации на основе прогресса.

3.3 Базы данных

При выборе базы данных важно учитывать различные факторы, а именно надежность, производительность, возможность масштабирования.

PostgreSQL [3] – это реляционная база данных с открытым исходным кодом, которая имеет отличную репутацию за свою надежность. PostgreSQL выделяется своей гибкостью, легкостью и низкими затратами на ресурсы.

В таблице 5 представлена сравнительная характеристика популярных СУБД.

Таблица 5 – Сравнительная характеристика СУБД

Название СУБД	Преимущества	Недостатки
PostgreSQL	Надежность и гибкость, поддержка сложных SQL-запросов и транзакций, поддержка полнотекстового поиска и индексов на выражениях	Сложная настройка и администрирование, высокая требовательность к ресурсам
MySQL	Высокая скорость, простой в использовании	Менее строгая поддержка стандартов SQL
SQLite	Легковесная и удобная для малых проектов и прототипов, не требует установки отдельного сервера	Ограничена в плане масштабируемости, поддерживает только базовые SQL-запросы

С точки зрения выявленных требований и ограничений системы и разработанной архитектуры, PostgreSQL является лучшим вариантом, так как обеспечивает высокую скорость, надежность в работе.

3.4 Разворачивание

Docker [4] – платформа для контейнеризации, которая позволяет разработчикам упаковывать приложения и все их зависимости в изолированные контейнеры. Контейнеры представляют собой легковесные и независимые среды, содержащие код приложения, библиотеки, файлы конфигурации и другие зависимости, что делает их легкими для развертывания на различных системах. Контейнеры могут запускаться в изолированных пространствах на одном и том же сервере, благодаря чему достигается эффективное использование ресурсов.

Таким образом, после обзора и анализа доступных технологий, был выбран необходимый стек технологий, но основе которого сформирована диаграмма развертывания системы, изображённая на рисунке 3.1.

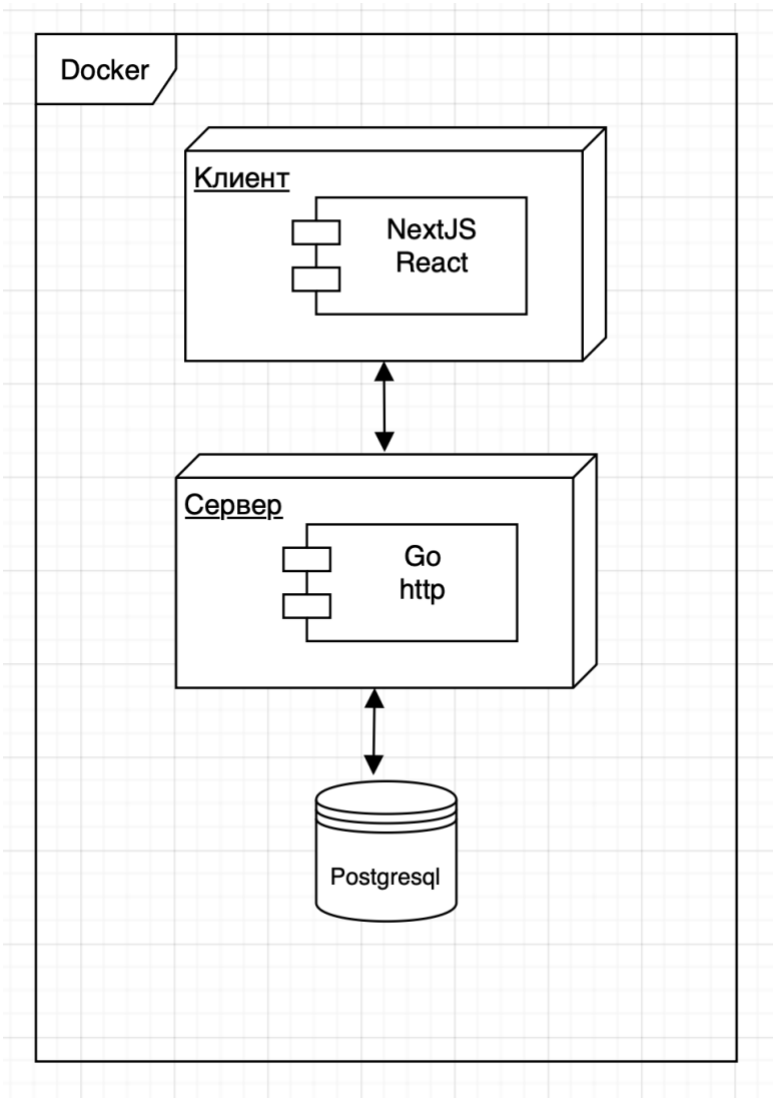


Рисунок 3.1 – Диаграмма развертывания системы

4 РАЗРАБОТКА И РАЗВОРАЧИВАНИЕ

4.1 Разработка серверной части

После анализа предметной области, выбора подходящей архитектуры и ее составления, описания компонентов системы и сценариев взаимодействия с системой, выбора необходимых технологий, необходимо приступить к непосредственной реализации системы.

Начиная с серверной части приложения, были реализованы основные точки взаимодействия с серверной частью. приложения.

На рисунках 4.1-4.4 представлена основная логика серверной части приложения.

```
type UserHandler struct {
    *configs.Config
    UserRepository *UserRepository
}

func NewUserHandler(router *http.ServeMux, deps UserHandler) {
    handler := deps
    router.HandleFunc("GET /user", handler.Auth())
    router.HandleFunc("POST /user/description", handler.Description())
    router.HandleFunc("GET /user/leaderboard", handler.Leaderboard())
    router.HandleFunc("POST /user/click", handler.Click())
    router.HandleFunc("GET /user/cost", handler.Cost())
    router.HandleFunc("GET /user/auto", handler.Auto())
}

// Получение информции о юзере + автоизация
func (handler *UserHandler) Auth() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        body, err := req.HandleBody[UserRequest](w, r)
        if err != nil {
            return
        }
        userTemp := &User{
            TelegramID: body.TelegramID,
            Name:       body.Name,
        }
        user, err := handler.UserRepository.Create(userTemp)
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }
        response := UserResponse{
            Balance:     user.Balance,
            MaxTokens:   user.MaxTokens,
            Description: user.Description,
        }
        res.Json(w, response, 200)
    }
}

// Запрос на изменение описания
func (handler *UserHandler) Description() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        body, err := req.HandleBody[UserDescriptionRequest](w, r)
        if err != nil {
            return
        }
        user, err := handler.UserRepository.ChangeDescription(body.TelegramID, body.Description)
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }
        res.Json(w, user, 200)
    }
}

// Получение топа
func (handler *UserHandler) Leaderboard() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        users, err := handler.UserRepository.GetTopByBalance()
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }
```

Рисунок 4.1 – Обработчик запросов работы с пользователем

```go
type StoreHandler struct {
	*configs.Config
	StoreRepository *StoreRepository
}

func NewStoreHandler(router *http.ServeMux, deps StoreHandler) {
	handler := deps
	router.HandleFunc("GET /store/clicks", handler.StoreClick())
	router.HandleFunc("GET /store/auto", handler.StoreAuto())
	router.HandleFunc("GET /store/buy", handler.Buy())
}

func (handler *StoreHandler) StoreClick() http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		body, err := req.HandleBody[StoreRequest](w, r)
		if err != nil {
			return
		}
		items, err := handler.StoreRepository.GetClicksStore(body.TelegramID)
		if err != nil {
			http.Error(w, err.Error(), http.StatusBadRequest)
			return
		}
		res.Json(w, items, 200)
	}
}

func (handler *StoreHandler) StoreAuto() http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		body, err := req.HandleBody[StoreRequest](w, r)
		if err != nil {
			return
		}
		items, err := handler.StoreRepository.GetAutoStore(body.TelegramID)
		if err != nil {
			http.Error(w, err.Error(), http.StatusBadRequest)
			return
		}
		res.Json(w, items, 200)
	}
}

func (handler *StoreHandler) Buy() http.HandlerFunc {
	return func(w http.ResponseWriter, r *http.Request) {
		body, err := req.HandleBody[StoreBuyRequest](w, r)
		if err != nil {
			return
		}
		err = handler.StoreRepository.Buy(body.TelegramID, body.Store, body.Name)
		if err != nil {
			http.Error(w, err.Error(), http.StatusBadRequest)
			return
		}
		res.Json(w, "Успешная покупка!", 200)
	}
}
```

Рисунок 4.2 – Обработчик запросов работы с магазином

```go
type Config struct {
	Db   DbConfig
	Auth AuthConfig
}

type DbConfig struct {
	DatabasePort     string
	DatabaseUser     string
	DatabasePassword string
	DatabaseName     string
}

type AuthConfig struct {
	Secret string
}

func LoadConfig() *Config {
	return &Config{
		Db: DbConfig{
			DatabasePort:     os.Getenv("DATABASE_PORT"),
			DatabaseUser:     os.Getenv("DATABASE_USER"),
			DatabasePassword: os.Getenv("DATABASE_PASSWORD"),
			DatabaseName:     os.Getenv("DATABASE_NAME"),
		},
		Auth: AuthConfig{
			Secret: os.Getenv("SECRET"),
		},
	}
}
```

Рисунок 4.3 – Конфигурационные данные

```
func App() http.Handler {
	conf := configs.LoadConfig()
	db := db.New(conf)

	go func() {
		ticker := time.NewTicker(time.Second)
		defer ticker.Stop()
		for range ticker.C {
			db.DB.Exec("SELECT increase_max_tokens()")
		}
	}()

	go func() {
		ticker := time.NewTicker(1 * time.Second)
		defer ticker.Stop()
		for range ticker.C {
			db.DB.Exec("SELECT add_auto_income_to_users()")
		}
	}()

	router := http.NewServeMux()

	userRepository := user.NewUserRepository(db)
	storeRepository := store.NewStoreRepository(db)

	user.NewUserHandler(router, user.UserHandler{
		Config:·····conf,
		UserRepository: userRepository,
	})

	store.NewStoreHandler(router, store.StoreHandler{
		Config:·····conf,
		StoreRepository: storeRepository,
	})

	stack := middleware.Chain(
		middleware.CORS,
	)
	return stack(router)
}

func main() {
	app := App()
	server := http.Server{
		Addr:····string(":" + os.Getenv("SERVER_PORT")),
		Handler: app,
	}
	fmt.Println("Server is listining on port", os.Getenv("SERVER_PORT"))
	server.ListenAndServe()
}
```

Рисунок 4.4 – Точка начала работы приложения

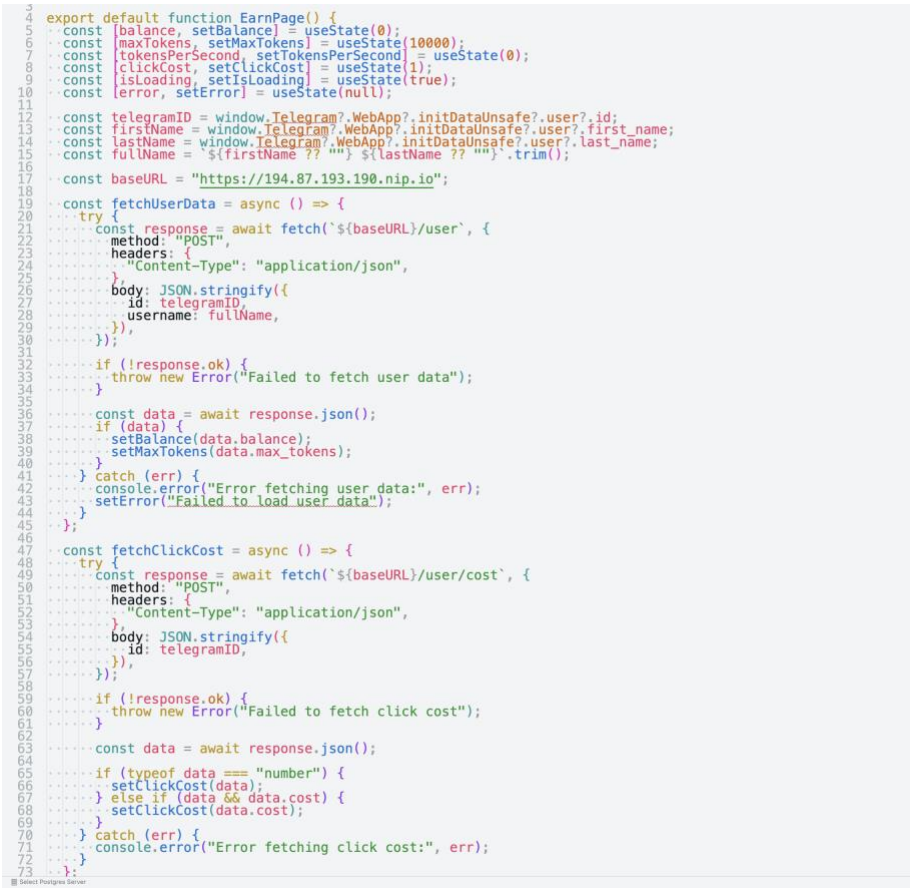
4.2 Разработка клиентской части

После реализации серверной части, обеспечивающей основные функции взаимодействия и хранения данных, следующим этапом стала разработка клиентской части приложения. Клиентская часть отвечает за отображение информации пользователю, обработку его действий, а также за отправку запросов к серверу через интерфейс.

На рисунках 4.5-4.8 представлена основная логика клиентской части приложения.

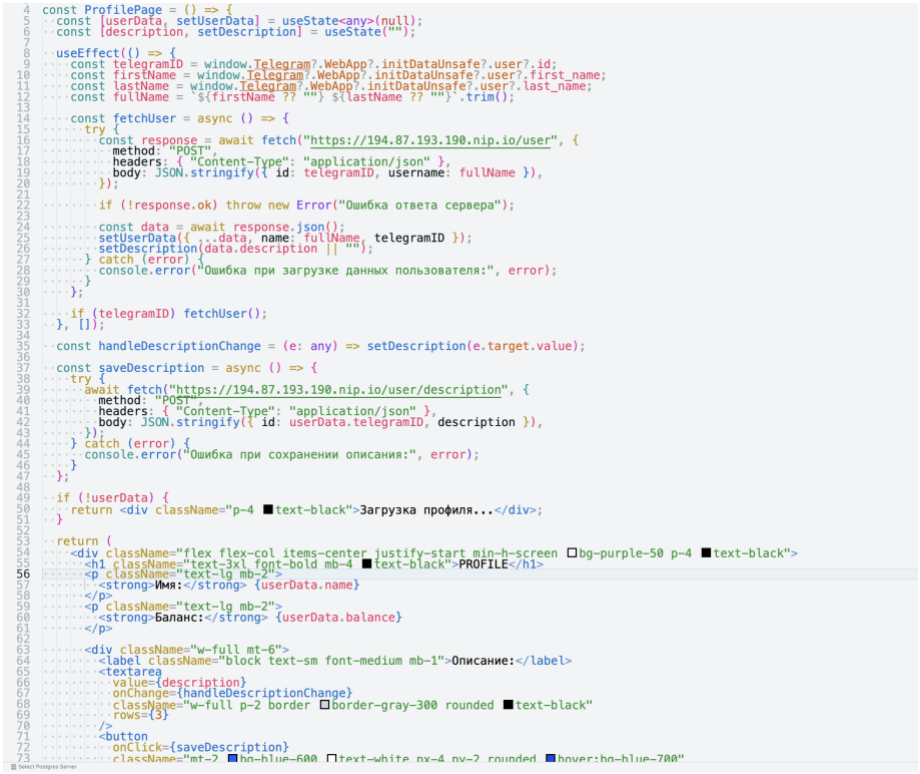
```
const LeaderboardPage = () => {
  const [leaderboardData, setLeaderboardData] = useState([]);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchLeaderboard = async () => {
      try {
        setIsLoading(true);
        const response = await fetch(
          "https://194.87.193.190.nip.io/user/leaderboard",
          {
            method: "GET",
            headers: {
              Accept: "application/json",
            },
          }
        );

        if (!response.ok) {
          throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const data = await response.json();
        setLeaderboardData(data);
      } catch (err) {
        console.error("Leaderboard fetch error:", err);
        setError(err.message || "Failed to load leaderboard");
      } finally {
        setIsLoading(false);
      }
    };

    fetchLeaderboard();
  }, []);

  return (
    <div className="flex flex-col h-screen bg-yellow-50 p-4">
      <h1 className="text-3xl font-bold mb-6 text-center flex items-center justify-center text-black">
        <Trophy className="mr-2" /> LEADERBOARD
      </h1>
      <h2 className="text-3xl font-bold mb-6 text-center flex items-center justify-center text-black">
        {" "}
        Кто первый заработает миллиард?
      </h2>

      {isLoading && (
        <div className="text-center py-4 text-lg text-gray-600">
          Загрузка данных...
        </div>
      )}

      {error && (
        <div className="text-center py-4 text-red-500">
          Ошибка загрузки: {error}
        </div>
      )}

      <div className="overflow-y-auto flex-grow">
        <div className="bg-white rounded-lg shadow-md">
          {leaderboardData.length > 0 ? (
            leaderboardData.map((user, index) => (
              <div
                key={user.TelegramID}
                className={`flex items-center justify-between p-4 ${
                  index !== leaderboardData.length - 1
                    ? "border-b border-gray-200"
                    : ""
                } mb-2`}
              >
```

Рисунок 4.5 – Отображение топа игроков



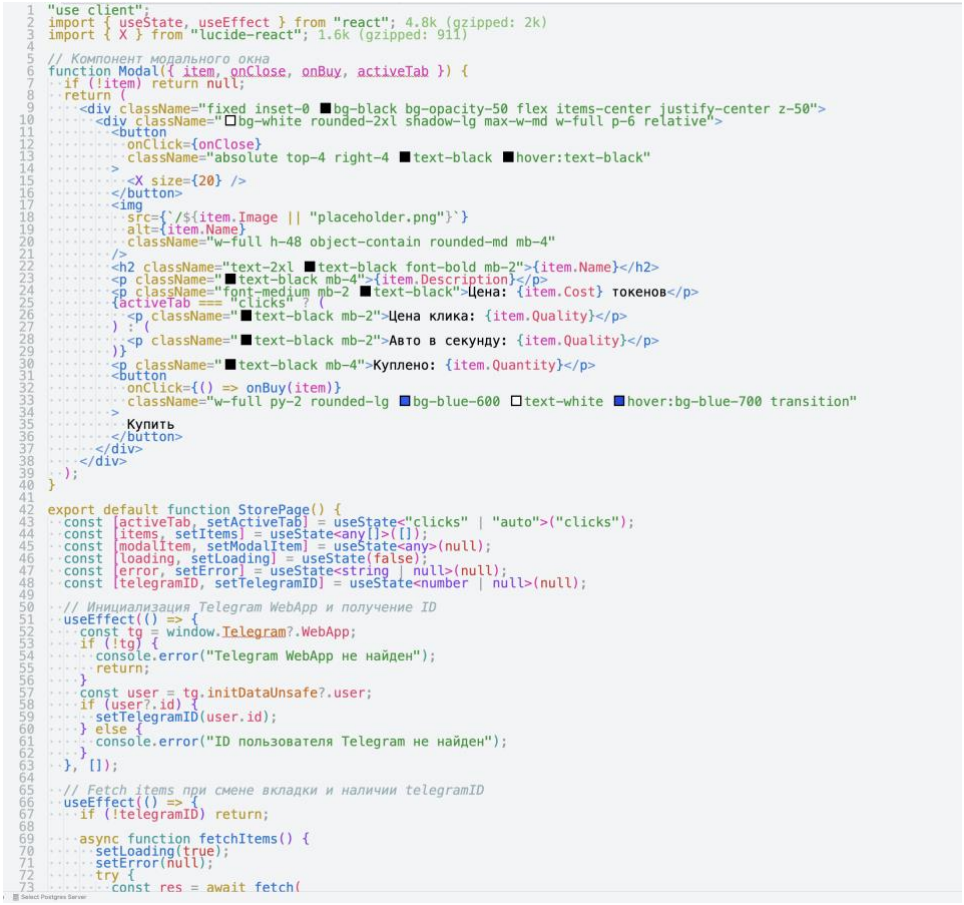
```
export default function EarnPage() {
  const [balance, setBalance] = useState(0);
  const [maxTokens, setMaxTokens] = useState(10000);
  const [tokensPerSecond, setTokensPerSecond] = useState(0);
  const [clickCost, setClickCost] = useState(1);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);

  const telegramID = window.Telegram?.WebApp?.initDataUnsafe?.user?.id;
  const firstName = window.Telegram?.WebApp?.initDataUnsafe?.user?.first_name;
  const lastName = window.Telegram?.WebApp?.initDataUnsafe?.user?.last_name;
  const fullName = `${firstName ?? ""} ${lastName ?? ""}`.trim();

  const baseURL = "https://194.87.193.190.nip.io";

  const fetchUserData = async () => {
    try {
      const response = await fetch(`${baseURL}/user`, {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          id: telegramID,
          username: fullName,
        }),
      });

      if (!response.ok) {
        throw new Error("Failed to fetch user data");
      }

      const data = await response.json();
      if (data) {
        setBalance(data.balance);
        setMaxTokens(data.max_tokens);
      }
    } catch (err) {
      console.error("Error fetching user data:", err);
      setError("Failed to load user data");
    }
  };

  const fetchClickCost = async () => {
    try {
      const response = await fetch(`${baseURL}/user/cost`, {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          id: telegramID,
        }),
      });

      if (!response.ok) {
        throw new Error("Failed to fetch click cost");
      }

      const data = await response.json();

      if (typeof data === "number") {
        setClickCost(data);
      } else if (data && data.cost) {
        setClickCost(data.cost);
      }
    } catch (err) {
      console.error("Error fetching click cost:", err);
    }
  };
```

Рисунок 4.6 – Отображение заработка монет



```tsx
const ProfilePage = () => {
  const [userData, setUserData] = useState<any>(null);
  const [description, setDescription] = useState("");

  useEffect(() => {
    const telegramID = window.Telegram?.WebApp?.initDataUnsafe?.user?.id;
    const firstName = window.Telegram?.WebApp?.initDataUnsafe?.user?.first_name;
    const lastName = window.Telegram?.WebApp?.initDataUnsafe?.user?.last_name;
    const fullName = `${firstName ?? ""} ${lastName ?? ""}`.trim();

    const fetchUser = async () => {
      try {
        const response = await fetch("https://194.87.193.190.nip.io/user", {
          method: "POST",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify({ id: telegramID, username: fullName }),
        });

        if (!response.ok) throw new Error("Ошибка ответа сервера");

        const data = await response.json();
        setUserData({ ...data, name: fullName, telegramID });
        setDescription(data.description || "");
      } catch (error) {
        console.error("Ошибка при загрузке данных пользователя:", error);
      }
    };

    if (telegramID) fetchUser();
  }, []);

  const handleDescriptionChange = (e: any) => setDescription(e.target.value);

  const saveDescription = async () => {
    try {
      await fetch("https://194.87.193.190.nip.io/user/description", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ id: userData.telegramID, description }),
      });
    } catch (error) {
      console.error("Ошибка при сохранении описания:", error);
    }
  };

  if (!userData) {
    return <div className="p-4 text-black">Загрузка профиля...</div>;
  }

  return (
    <div className="flex flex-col items-center justify-start min-h-screen bg-purple-50 p-4 text-black">
      <h1 className="text-3xl font-bold mb-4 text-black">PROFILE</h1>
      <p className="text-lg mb-2">
        <strong>Имя:</strong> {userData.name}
      </p>
      <p className="text-lg mb-2">
        <strong>Баланс:</strong> {userData.balance}
      </p>

      <div className="w-full mt-6">
        <label className="block text-sm font-medium mb-1">Описание:</label>
        <textarea
          value={description}
          onChange={handleDescriptionChange}
          className="w-full p-2 border border-gray-300 rounded text-black"
          rows={3}
        />
        <button
          onClick={saveDescription}
          className="mt-2 bg-blue-600 text-white px-4 py-2 rounded hover:bg-blue-700"
```

Рисунок 4.7 – Отображение профиля игрока



```tsx
"use client";
import { useState, useEffect } from "react"; 4.8k (gzipped: 2k)
import { X } from "lucide-react"; 1.6k (gzipped: 911)

// Компонент модального окна
function Modal({ item, onClose, onBuy, activeTab }) {
  if (!item) return null;
  return (
    <div className="fixed inset-0 bg-black bg-opacity-50 flex items-center justify-center z-50">
      <div className="bg-white rounded-2xl shadow-lg max-w-md w-full p-6 relative">
        <button
          onClick={onClose}
          className="absolute top-4 right-4 text-black hover:text-black"
        >
          <X size={20} />
        </button>
        <img
          src={`/${item.Image || "placeholder.png"}`}
          alt={item.Name}
          className="w-full h-48 object-contain rounded-md mb-4"
        />
        <h2 className="text-2xl text-black font-bold mb-2">{item.Name}</h2>
        <p className="text-black mb-4">{item.Description}</p>
        <p className="font-medium mb-2 text-black">Цена: {item.Cost} токенов</p>
        {activeTab === "clicks" ? (
          <p className="text-black mb-2">Цена клика: {item.Quality}</p>
        ) : (
          <p className="text-black mb-2">Авто в секунду: {item.Quality}</p>
        )}
        <p className="text-black mb-4">Куплено: {item.Quantity}</p>
        <button
          onClick={() => onBuy(item)}
          className="w-full py-2 rounded-lg bg-blue-600 text-white hover:bg-blue-700 transition"
        >
          Купить
        </button>
      </div>
    </div>
  );
}

export default function StorePage() {
  const [activeTab, setActiveTab] = useState<"clicks" | "auto">("clicks");
  const [items, setItems] = useState<any[]>([]);
  const [modalItem, setModalItem] = useState<any>(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);
  const [telegramID, setTelegramID] = useState<number | null>(null);

  // Инициализация Telegram WebApp и получение ID
  useEffect(() => {
    const tg = window.Telegram?.WebApp;
    if (!tg) {
      console.error("Telegram WebApp не найден");
      return;
    }
    const user = tg.initDataUnsafe?.user;
    if (user?.id) {
      setTelegramID(user.id);
    } else {
      console.error("ID пользователя Telegram не найден");
    }
  }, []);

  // Fetch items при смене вкладки и наличии telegramID
  useEffect(() => {
    if (!telegramID) return;

    async function fetchItems() {
      setLoading(true);
      setError(null);
      try {
        const res = await fetch(
```

Рисунок 4.8 – Отображение магазина

4.3 Разворачивание

Чтобы развернуть разработанную систему необходимо прописать файл Dockerfile и docker-compose, представленные на рисунках 4.9 и 4.10.

```
FROM golang:1.24-alpine

WORKDIR /app

COPY go.mod ./
COPY go.sum ./
RUN go mod download

COPY . .


WORKDIR /app/cmd

RUN go build -o main

EXPOSE 8080

CMD ["./main"]
```

Рисунок 4.9 – Содержимое файла Dockerfile

```yaml
version: "3.8"

services:
  manager-service:
    build: .
    container_name: "game-service"
    expose:
      - "8080"
    environment:
      - DATABASE_PORT=5432
      - DATABASE_USER=postgres
      - DATABASE_PASSWORD=password
      - DATABASE_NAME=coin
      - DATABASE_HOST=db
      - SERVER_PORT=8080
      - SECRET=0hTBL4aYdp0EtnQS7LmQaxWWDZ3k3CRo/ibK1nOTnCM=
    depends_on:
      db:
        condition: service_healthy
    networks:
      - internal
      - web

  db:
    image: postgres:16
    container_name: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
      POSTGRES_DB: coin
    volumes:
      - ./migrations/init.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "5432:5432"
    healthcheck:
      test: ["CMD-SHELL", "sh -c 'pg_isready -U postgres -d coin'"]
      interval: 5s
      timeout: 10s
      retries: 5
      start_period: 10s
    networks:
      - internal

  nginx:
    image: nginx:latest
    container_name: nginx
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - ./certbot/www:/var/www/certbot
      - ./certbot/conf:/etc/letsencrypt
    depends_on:
      - manager-service
    networks:
      - web

  certbot:
    image: certbot/certbot
    container_name: certbot
    volumes:
      - ./certbot/www:/var/www/certbot
      - ./certbot/conf:/etc/letsencrypt
    entrypoint: "/bin/sh -c"
    command: >
      "sleep 5 &&
```

Рисунок 4.10 – Содержимое файла docker-compose

На рисунке 4.11 представлена конфигурация веб-сервера Nginx.

```
events {}

http {
    server {
        listen 80;
        server_name 194.87.193.190.nip.io;

        location /.well-known/acme-challenge/ {
            root /var/www/certbot;
        }

        location / {
            return 301 https://$host$request_uri;
        }
    }

    server {
        listen 443 ssl;
        server_name 194.87.193.190.nip.io;

        ssl_certificate /etc/letsencrypt/live/194.87.193.190.nip.io/fullchain.pem;
        ssl_certificate_key /etc/letsencrypt/live/194.87.193.190.nip.io/privkey.pem;

        location / {
            proxy_pass http://manager-service:8080;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
}
```

Рисунок 4.11 – Содержимое файла nginx.conf

4.3 Тестирование

Также проведем ручные тесты на работоспособность приложения. Тестирование представлено на рисунках с 4.12 по 4.16, в течение которого проводилось тестирования доступного функционала приложения со стороны клиента.

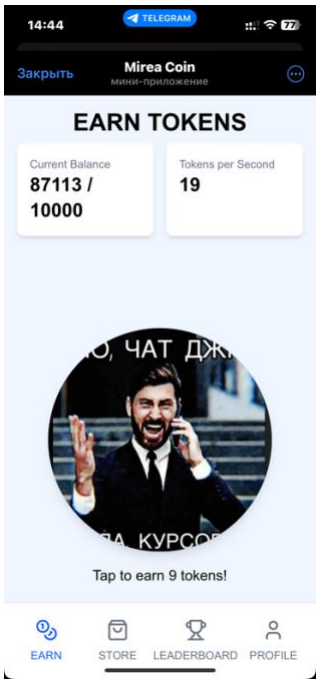


Рисунок 4.12 – Тестирование входа в игру

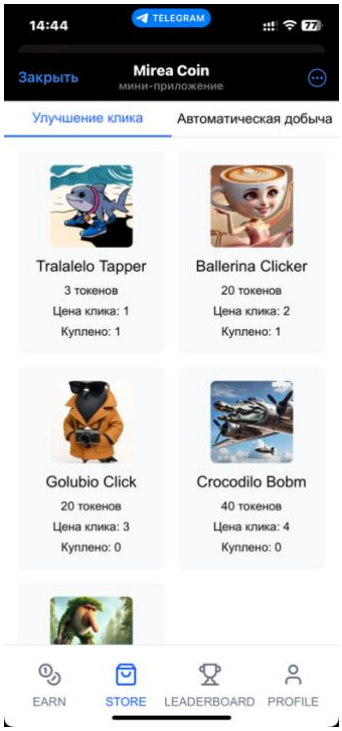


Рисунок 4.13 – Тестирование открытия магазина

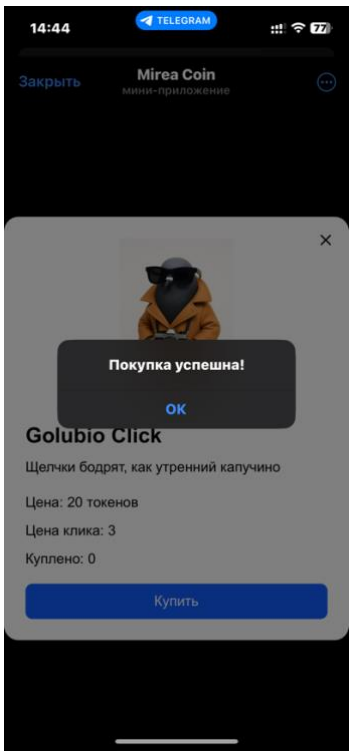


Рисунок 4.14 – Тестирование покупки предмета

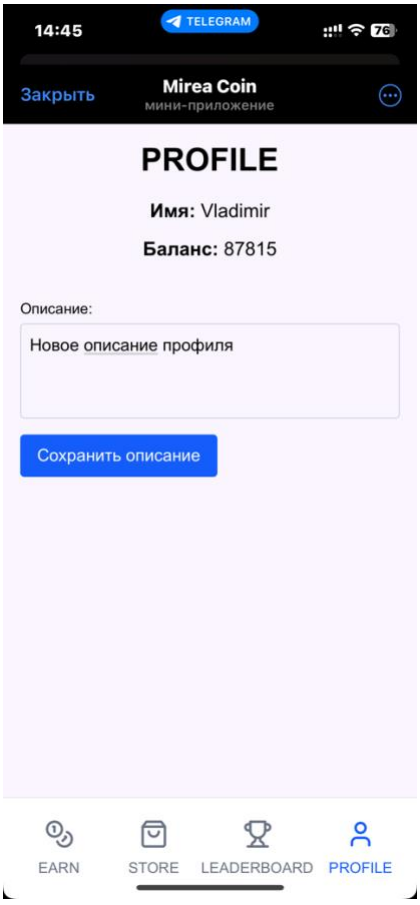


Рисунок 4.15 – Тестирование изменения описания профиля

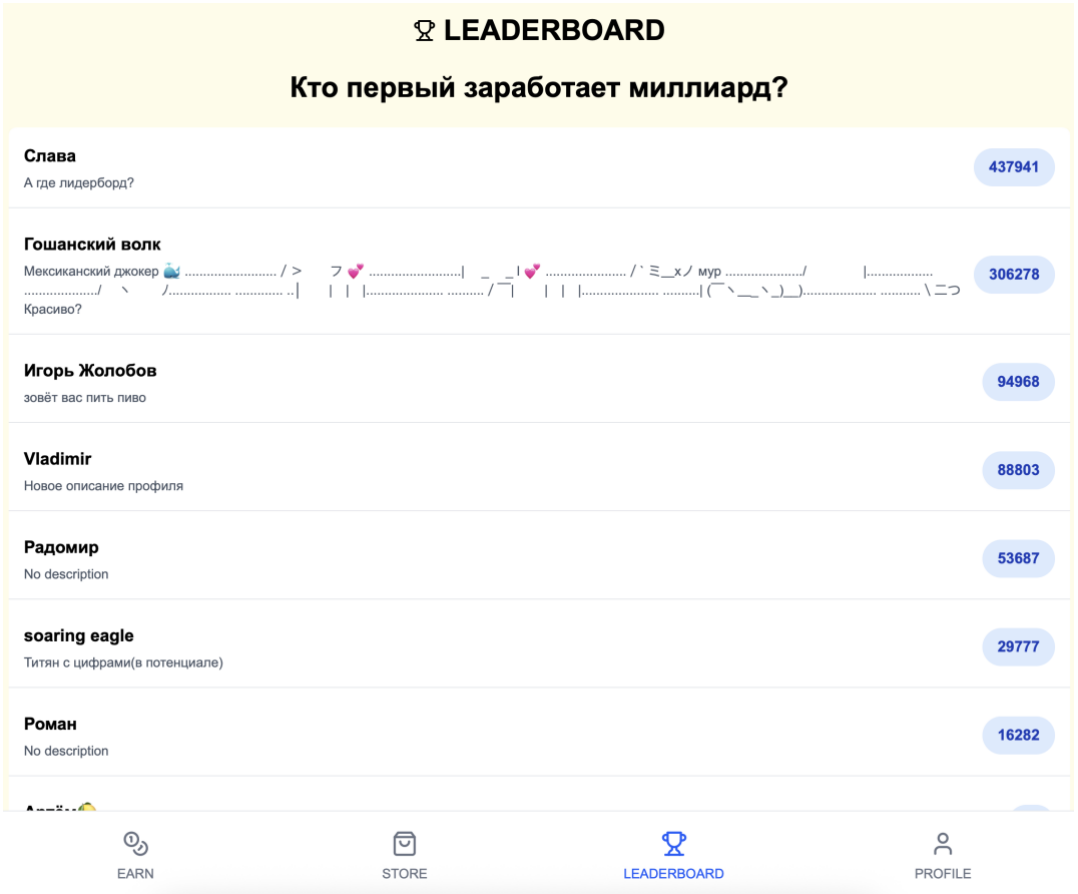


Рисунок 4.16 – Тестирование получения информации топа

Для проведения фаззинг-тестирования API была использована библиотека Schemathesis, предназначенная для автоматической генерации запросов на основе спецификации OpenAPI. В ходе тестирования выполнялась отправка различных корректных и некорректных запросов с целью выявления возможных уязвимостей и нестабильной работы сервера при обработке нестандартных входных данных. Проведение тестов позволило убедиться в корректной обработке большинства запросов со стороны сервиса. Результаты фаззинг-тестирования представлены на рисунке 4.17.12

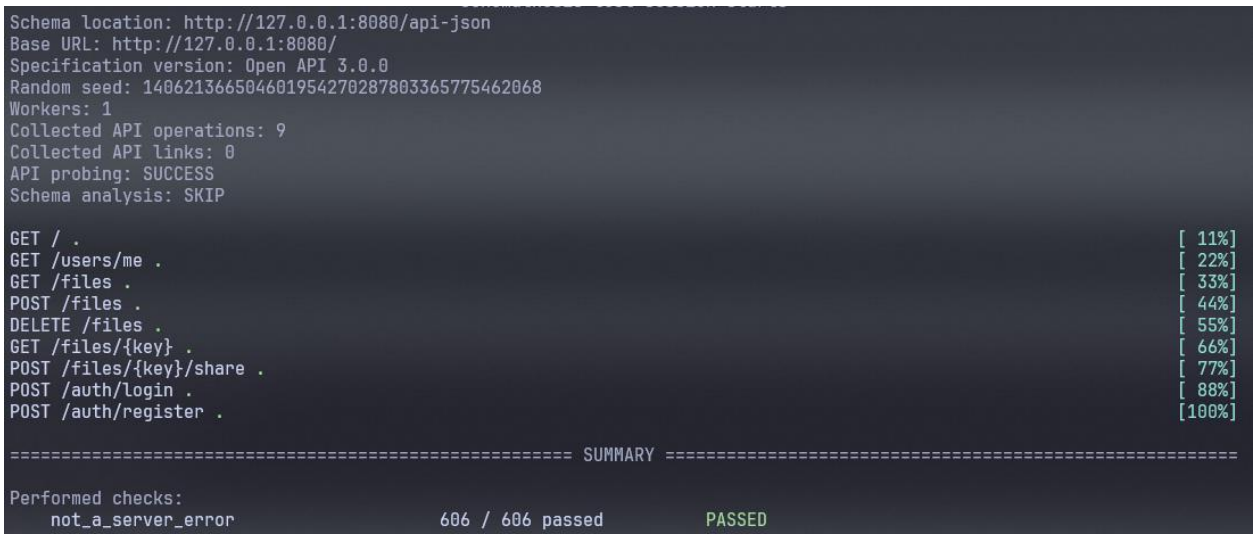


Рисунок 4.17 – Результат фаззинг-тестирования

ЗАКЛЮЧЕНИЕ

В результате разработки приложения была выполнена цель и поставленные задачи, проанализированы предметная область, выделены функциональные и нефункциональные требования, ограничения к системе, проанализирован программный инструментарий, разработана серверная и клиентская часть приложения с использованием выбранной архитектуры и средств разработки, проведено тестирование ручным методом.

Разработанная серверная и клиентская часть приложения работают стабильно и предоставляет необходимый и работоспособный функционал для пользователей.

Разработанная серверная и клиентская часть приложения были развернуты на удаленном сервере. Тесты показали положительную оценку с точки зрения корректности использования приложения.

Для ознакомления разработанным приложением, представлены ссылки на программный код данного приложения – https://github.com/iqohgog/mirea-coin, и на хостинг клиентской части – https://t.me/mireacoinbot.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. NextJS Docs / [Электронный ресурс] // nextjs.org : [сайт]. — URL: https://nextjs.org/docs (дата обращения: 01.05.2025).

2. React Docs / [Электронный ресурс] // react.dev : [сайт]. — URL: https://react.dev/reference/react (дата обращения: 01.05.2025).

3. Documentation / [Электронный ресурс] // postgresql.org : [сайт]. — URL: https://www.postgresql.org/docs/ (дата обращения: 01.04.2025).

4. Docker Manuals / [Электронный ресурс] // https://docs.docker.com/manuals/ : [сайт]. — URL: https://docs.docker.com (дата обращения: 01.04.2025).

5. Notcoin / [Электронный ресурс] // notcoin.org : [сайт]. — URL: https://notcoin.org (дата обращения: 10.05.2025).

6. HamsterKobmat / [Электронный ресурс] // hamsterkombatgame.io: [сайт]. — URL: https://hamsterkombatgame.io (дата обращения: 10.04.2024).

7. Cookie clicker / [Электронный ресурс] // cookieclicker.com : [сайт]. — URL: https://cookieclicker.com (дата обращения: 10.04.2025).

8. MVC / [Электронный ресурс] // blog.skillfactory.ru : [сайт]. — URL: https://blog.skillfactory.ru/glossary/mvc/ (дата обращения: 13.05.2025).

9. Клиент-серверная архитектура в картинках / [Электронный ресурс] // habr.com : [сайт]. — URL: https://habr.com/ru/articles/495698 (дата обращения: 13.05.2025).

10. Основные архитектурные шаблоны построения ПО / [Электронный ресурс] // habr.com : [сайт]. — URL: https://habr.com/ru/companies/ruvds/articles/699648/ (дата обращения: 13.05.2025).

11. Просто о микросервисах / [Электронный ресурс] // habr.com : [сайт]. — URL: https://habr.com/ru/companies/raiffeisenbank/articles/346380 (дата обращения: 13.05.2025).

12. Build simple, secure, scalable systems with Go / [Электронный ресурс] // go.dev : [сайт]. — URL: https://go.dev/# (дата обращения: 14.05.2025).