МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
Факультет информационных технологий и робототехники

Кафедра программного обеспечения информационных систем и технологий

КУРСОВОЙ ПРОЕКТ

по дисциплине: «Компьютерные системы и сети»

на тему: «Сетевое ПО «Банковское приложение»»

Выполнил**:** ст. гр. 10701323 Шаплавский Н.С.

Приняла**:** ст. пр. Белова С.В.

Минск 2025

Оглавление

ВВЕДЕНИЕ

В современных условиях стремительного развития информационных технологий эффективное управление банковскими сервисами невозможно без внедрения специализированного программного обеспечения. Приложение банкинг как ключевой элемент финансовых услуг, требует автоматизации процессов обработки платежей, учета клиентов, управления транзакциями, контроля доступа к данным и обеспечения информационной безопасности.

Банковские приложения становятся важнейшим инструментом для оптимизации работы финансовых учреждений, повышения удобства пользователей и конкурентоспособности банка в целом. Целью данного курсового проекта является разработка концепции банковского приложения, адаптированного под потребности клиентов, с учетом особенностей его инфраструктуры и бизнес-процессов.

В работе рассматриваются вопросы интеграции платежных систем, модулей управления счетами, инструментов мониторинга транзакционной активности и защиты данных. Особое внимание уделяется проектированию архитектуры, выбору технологий и протоколов, обеспечивающих стабильность, масштабируемость и безопасность решения.

Курсовой проект актуален, потому что сегодня практически все финансовые операции переходят в цифровое пространство, и банки — не исключение. Современным финансовым учреждениям необходимо не только оперативно обрабатывать платежи и переводы, но и надёжно хранить данные клиентов, контролировать доступ к счетам, а также предотвращать ошибки и мошенничество.

1 Обзор состояния вопроса

1.1. Описание и анализ предметной области

Банковское программное обеспечение предназначено для упрощения и автоматизации банковских операций, таких как управление счетами, переводы, оплата услуг и мониторинг финансового состояния пользователя. При авторизации клиент вводит свои учетные данные, которые проверяются через систему безопасности. После успешной идентификации пользователю предоставляется доступ к его финансовой информации, включая баланс и доступные услуги.

Каждая операция в системе фиксируется в базе данных, обеспечивая безопасность и защиту от несанкционированного доступа. Пользователь может проводить финансовые операции, такие как переводы между своими счетами а так же другим пользователям. Все транзакции проходят через систему верификации и шифруются для защиты персональных данных.

В системе предусмотрены разграничения прав доступа: обычные пользователи могут управлять своими счетами и картами, сотрудники банка могут просматривать клиентские заявки и выполнять операции согласно своим полномочиям, а администраторы управляют пользователями и настройками безопасности.

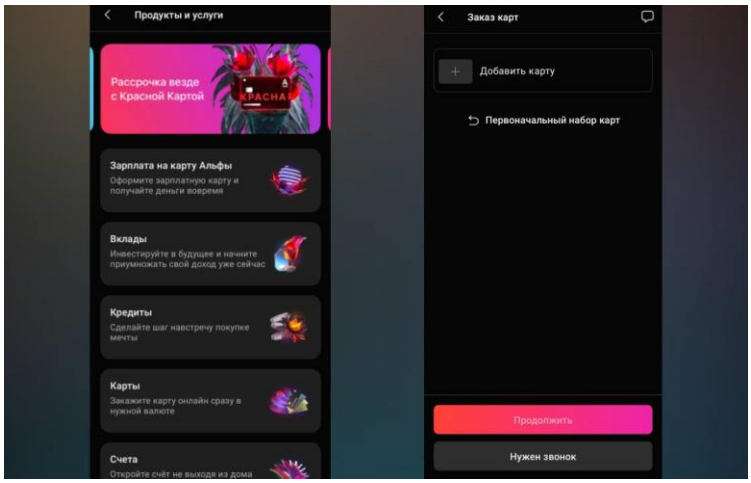
Разработка мобильного банкинга в рамках курсового проекта фокусируется на базовых функциях, таких как управление балансом, переводы, защита данных и удобный интерфейс. Для реализации используется **графический интерфейс на CTkinter(Python)**, серверная часть написана на **Python с использованием socket**, а база данных **PostgreSQL** обеспечивает надежное хранение информации. Для безопасной работы применяются **шифрование данных** и **протоколы аутентификации**, что гарантирует стабильность и защиту пользовательских данных.

1.2. Обзор аналогов

В ходе исследования были рассмотрены три популярных мобильных банковских приложения: Айсинк (Альфа-Банк Беларусь), Беларусьбанк, Тинькофф.

Айсинк (Альфа-Банк Беларусь)

Приложение от Альфа-Банка Беларусь с современным и удобным интерфейсом. Отличается возможностью открытия счетов онлайн, управления картами и кредитами, оплаты коммунальных услуг и перевода денег между банками. В Айсинк встроен чат-бот, который помогает клиентам решать вопросы без необходимости звонить в поддержку. Также есть система бонусов и кэшбэка. Однако приложение доступно только для клиентов Альфа-Банка и не поддерживает мультивалютные счета.

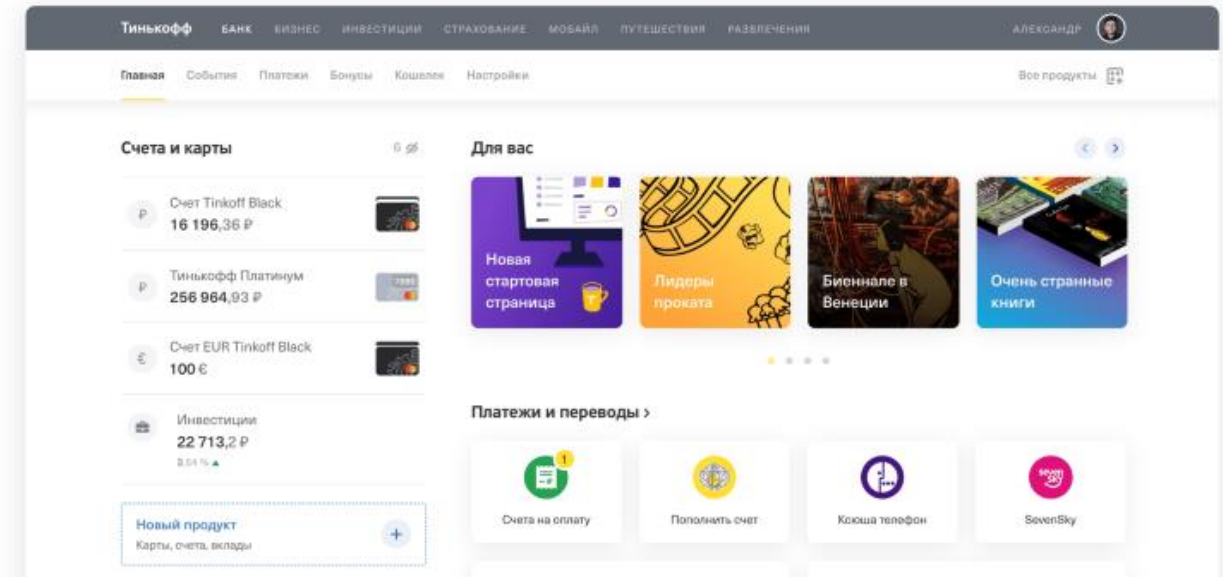


Беларусбанк

Официальное приложение крупнейшего государственного банка Беларуси. Оно предоставляет доступ к депозитам, кредитам, переводам, оплате услуг и обмену валют. Важным преимуществом является широкая сеть банкоматов и отделений, а также интеграция с ЕРИП (единая система платежей в Беларуси). Однако пользователи отмечают, что интерфейс приложения менее интуитивен, чем у частных банков, и иногда возникают технические сбои.

Тинькофф

Одно из самых продвинутых банковских приложений, которое работает полностью в цифровом формате. В нем есть гибкие настройки карт, инвестиции, накопительные счета, управление подписками, кредиты и страховки. Приложение автоматически анализирует расходы, предлагает удобные финансовые категории и имеет интеграцию с Apple Pay и Google Pay. Однако оно ориентировано на российский рынок и не поддерживает все функции за пределами России.



1.3Изучение используемых технологий и алгоритмов

При разработке клиент-серверного приложения «Банкинг» основной упор сделан на сокетное взаимодействие между клиентом и сервером, реализованным на языке Python. Это обеспечивает прямое двустороннее соединение и позволяет передавать данные в режиме реального времени без использования HTTP-протокола.

Серверная часть будет реализована с использованием стандартных средств Python (модуля socket) и поддерживать многопоточную архитектуру. Каждый клиент подключается к серверу через отдельный поток или процесс, что обеспечивает параллельную обработку запросов и предотвращает блокировку основного цикла приложения.

Для хранения данных используется СУБД PostgreSQL. Взаимодействие с базой данных организовано через ORM-библиотеки (SQLAlchemy), что упрощает работу с таблицами и автоматизирует выполнение SQL-запросов, а так же делает позволяет сделать транзакции атомарными. В структуру базы данных входят таблицы, отражающие основные сущности приложения: пользователи, банковские счета, транзакции, заявки и т.п.

Клиентское приложение также написано на Python, и при запуске оно устанавливает сокет-соединение с сервером. Для обеспечения отзывчивости интерфейса и избежания блокировки пользовательского интерфейса сетевое взаимодействие выносится в отдельные потоки.

Обмен данными между клиентом и сервером осуществляется в формате JSON, что делает структуру сообщений удобной для парсинга и расширяемой в будущем.

Такой подход обеспечивает гибкость, масштабируемость и прямой контроль над сетевым взаимодействием, что особенно важно для приложений в банковской сфере, где критически важны скорость и надежность передачи данных.

2. Разработка требований к программному обеспечению

Целью настоящего курсового проекта является создание удобной, надёжной и масштабируемой банковской системы, реализованной с использованием технологий Python, сокетного взаимодействия и СУБД PostgreSQL. Такая система должна автоматизировать ключевые финансовые процессы, повысить безопасность хранения и передачи данных, а также обеспечить эффективное взаимодействие между клиентами и сотрудниками банка.

Основными задачами разработки являются:

- обеспечение безопасного и стабильного обмена данными в режиме реального времени между клиентом и сервером;
- организация учёта клиентов, счетов и транзакций;
- реализация гибкой модели прав доступа;
- минимизация человеческого фактора и ускорение обработки запросов.

На основании этих целей формируются функциональные и нефункциональные требования.

2.1 Ролевое распределение

Приложение предусматривает три основные роли:

- **Администратор** — ведет контроль за стабильной работой приложения, имеет полный доступ к бд, для исправления ошибок, а так же проведеня расследований
- **Пользователь** — имеет ограниченный доступ: просмотр чужих профилей, контроль баланса и истории действий.

Роль Администратор:

1. Управление клиентами:
    - Регистрация: ФИО, email, паспортные данные.
2. Управление банковскими счетами:
    - Создание новых счетов для клиентов.
    - Просмотр баланса и операций по счёту.
    - Изменение статуса счета: активен, заблокирован, закрыт.
3. Работа с транзакциями:

- Создание и проведение перевода между счетами.
- Просмотр истории операций.

4. Автоматизация:
   - Автоматическое изменение статуса счёта после завершения действия (например, после перевода).
   - Ведение журнала всех операций.

*Роль **Пользователь**:*

1. Просмотр собственного профиля:
   - Информация: ФИО, паспорт, Номер телефона.
   - Свои активные счета, их балланс
2. Действия
   - Создание/удаление счета.
   - Перевод денег с одного счета на другой

2.2 Нефункциональные требования

1. Производительность и надёжность:

- Время отклика при передаче данных не должно превышать 2 секунд.
- Система должна работать стабильно 24/7.

2. Безопасность:

- Разграничение прав доступа: пользователя не могут видеть чужую информацию и проводить действия с чужими счетами.
- Хранение паролей и личных данных в хешированном виде.
- Шифрование всех передаваемых по сокетам данных.

3. Удобство использования:

- Простой и понятный интерфейс
- Основные разделы: сотрудники, клиенты, счета, транзакции.
- Четкие названия всех кнопок и действий, соответствие логике бизнес-процессов.

4. Используемые технологии:

- Клиентская часть: Python с GUI (CTK).
- Серверная часть: Python + socket, многопоточность

- База данных: PostgreSQL.
- Формат передачи данных: JSON.
- Протокол: TCP/IP.

3 Проектирование ПО

3.1 Описание вариантов использования

В разрабатываемом приложении «Банковское приложение» предусмотрены разнообразные возможности, направленные на обеспечение удобного доступа и управления своими счетами и деньгами на них.

Основными лицами в приложении являются «Пользователь» и «Администратор».

Полный перечень вариантов использования для «Пользователя» представлен на рисунке 3.1.

Рисунок 3.1 – Диаграмма вариантов использования «Пользователя»

Полный перечень вариантов использования для «Администратора» представлен на рисунке 3.2.

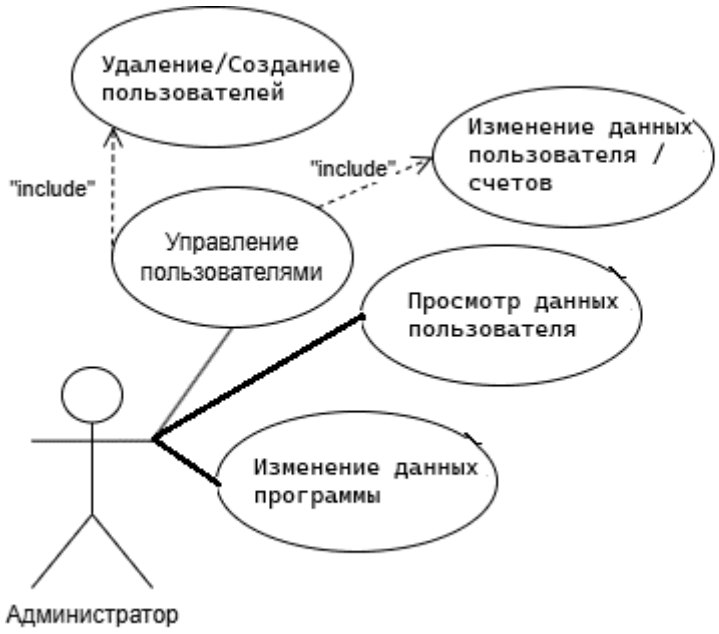


Рисунок 3.2 – Диаграмма вариантов использования «Администратора»

3.2 Архитектура ПО. Описание клиентской и серверной части

Программная архитектура представляет собой систему базовых принципов, определяющих организацию компонентов, их взаимосвязи и механизмы интеграции для достижения целевых показателей. В рамках проекта реализована клиент-серверная модель, обеспечивающая четкое разделение функциональности между серверной и клиентской частями, что оптимизирует процессы обработки данных и распределения вычислительной нагрузки.

Структурные компоненты системы:

1. Серверный модуль (Backend)

- Ядро бизнес-логики: обрабатывает запросы, управляет взаимодействием с СУБД PostgreSQL, обеспечивает целостность данных
- Сетевой контроллер: реализует TCP-коммуникацию, управляет пулом соединений, обеспечивает маршрутизацию запросов

2. Клиентский модуль (Frontend)

- Коммуникационный обработчик: поддерживает устойчивое соединение с сервером, сериализует/десериализует JSON-сообщения
- Интерфейсный блок: визуализирует банковские данные (балансы, реквизиты), реализует элементы управления

Механизм взаимодействия:

1. Установление соединения:

- Клиент инициирует TCP-сессию через сетевой модуль
- Сервер, используя механизм пулинга потоков, создает выделенный обработчик

2. Цикл обработки запросов:

- Клиент формирует JSON-запрос с указанием целевого действия
- Сервер валидирует запрос, исполняет через ядро бизнес-логики
- Результат возвращается структурированным JSON-ответом

3. Синхронизация состояния:

- Клиент периодически запрашивает данные, чтобы они оставались актуальны

Ключевые преимущества решения:

1. Гибкость системы:

- Независимое развитие серверной и клиентской компонент
- Возможность горизонтального масштабирования

2. Безопасность:

- Изоляция критических операций на серверной стороне
- Контролируемые точки входа/выхода данных

3. Производительность:

- Оптимизированная работа с СУБД через подготовленные запросы
- Асинхронная обработка соединений

4. Удобство сопровождения:

- Четкое разделение зон ответственности
- Стандартизированные протоколы взаимодействия

Реализованная архитектура обеспечивает эффективное управление финансовыми операциями, сочетая высокую производительность обработки транзакций с удобством пользовательского интерфейса, что соответствует современным требованиям к банковским информационным системам.

3.3 Описание протокола взаимодействия клиента и сервера. Структура передаваемых сообщений

Протокол взаимодействия клиента и сервера описывает правила и формат обмена сообщениями между клиентским и серверным приложениями. Протокол включает в себя структуру передаваемых сообщений, последовательность взаимодействия и типы поддерживаемых команд.

Структура передаваемых сообщений

Сообщения между клиентом и сервером передаются в формате текстовых строк. Для передачи данных о процессах используется формат JSON.

Взаимодействие клиента и сервера приведены ниже в таблице.

Имя команды	Формат команды	Описание	Ответ сервера

Currency_api	{
	‘headers’:{
	'method': 'get',
	'route':'currency_api',
	'JWT': ТОКЕН_КЛИЕНТА,
	
	"ip":IP_АДРЕСС_КЛИЕНТА,
	"config_version":
	ВЕРСИЯ_КОНФИГА
	}
	‘data’:{
	None
	}
	}
		Команда для получения курса валют	Возвращает курсы валют

login	{
	‘headers’:{
	'method': 'post',
	'route': 'login',
	'JWT': None,
	
	"ip":IP_АДРЕСС_КЛИЕНТА,
	"config_version":
	ВЕРСИЯ_КОНФИГА
	}
	‘data’:{
	“telephone”:НОМЕР,
	“password”: ПАРОЛЬ,
	}
	}
		Команда для авторизации пользователя	Верифицирует пользователя, и если все верно, возвращает ему токен авторизации

registration	{
	‘headers’:{
	'method': 'post',
	'route':
	'registration',
	'JWT': None,
	
	"ip":IP_АДРЕСС_КЛИЕНТА,
	"config_version":
	ВЕРСИЯ_КОНФИГА
	}
	‘data’:{
	“name”: Имя,
	“surname”:Фамилия,
	“passport_number”:ном
	паспорта,
	“passport_id”:инд номер,
	“telephone”:Номер тел,
	“password”:Пароль,
	
	}
	}
		Команда для регистрации	Регистрирует пользователя и отправляет ему токен авторизации

Check auth	{
	'headers':{
	'method': 'post',
	'route':
	'check_auth',
	'JWT': ТОКЕН_КЛИЕНТА,
	
	"ip":IP_АДРЕСС_КЛИЕНТА,
	"config_version":
	ВЕРСИЯ_КОНФИГА
	}
	'data':{
	None,
	}
	}
		Команда проверки аунтефикации	Сервер проверяет авторизацию и возвращает пользователю статут(авторизован или нет), так же новый токен авторизации и данные пользователя

Get user data api	{
	'headers':{
	'method': 'post',
	'route':
	'get_user_data_api',
	'JWT': ТОКЕН_КЛИЕНТА,
	
	"ip":IP_АДРЕСС_КЛИЕНТА,
	"config_version":
	ВЕРСИЯ_КОНФИГА
	}
	'data':{
	None,
	}
	}
		Для получения всех данных о клиенте	Сервер проверяет авторизацию и возвращает все данные о пользователе (его счета и т.д.)

Delete card api	{
	'headers':{
	'method': 'post',
	'route':
	'delete_card_api',
	'JWT': ТОКЕН_КЛИЕНТА,
	
	"ip":IP_АДРЕСС_КЛИЕНТА,
	"config_version":
	ВЕРСИЯ_КОНФИГА
	}
	'data':{
	"card_number":Номер
	карты,
	}
	}
		Команда для удаления счета	Возвращает статус (Удалена/ошибка удаления т.к. …)

Create product	'headers':{ 'method': 'post', 'route': 'create_cproduct_api', 'JWT': ТОКЕН_КЛИЕНТА, "ip":IP_АДРЕСС_КЛИЕНТА, "config_version":	Команда для создания нового счета	Возвращает статус (Создан/ошибка создания т.к. …)

	ВЕРСИЯ_КОНФИГА
	}
	'data':{
	"":"",
	"product_type":Тип,
	"is_named_product":имя,
	"currency":Валюта,
	
	}
	}		

Transfer money	'headers':{
	'method': 'post',
	'route':
	'transfer_money_api',
	'JWT': ТОКЕН_КЛИЕНТА,
	
	"ip":IP_АДРЕСС_КЛИЕНТА,
	"config_version":ВЕРСИЯ_КОНФИГА
	}
	'data':{
	"adr": Куда;
	"card_number":От куда,
	"transfer_type":тип,
	"sum":сумма,
	}
	}
		Команда для перевода денег	Возвращает статус (Переведено/ошибка перевода т.к. …)

Последовательность взаимодействия

1) Установление соединения: Клиент устанавливает TCP-соединение с сервером. Сервер принимает подключение и создаёт отдельный поток для обработки запросов данного клиента.

2) Передача данных: Запросы от клиента и ответы сервера оформляются в формате JSON, что позволяет легко сериализовать и десериализовать данные.

3) Обработка команд: После получения запроса сервер выполняет соответствующие действия, обращается к базе данных (PostgreSQL) при необходимости и возвращает клиенту результат операции.

4) Динамическое обновление: Сервер переодически отправляет данные клиенту, чтобы данные которые он видит были актуальны.

Данная архитектура и протокол взаимодействия обеспечивают гибкость, безопасность и масштабируемость системы, что позволяет удовлетворить функциональные требования как для конечного пользователя, так и для администратора, а также обеспечить эффективное управление содержимым электронной библиотеки.

3.4 Проектирование интерфейса пользователя

Проектирование интерфейса пользователя направлено на создание интуитивно понятного взаимодействия пользователей с программой. Основная цель интерфейса – предоставить пользователям доступ ко своим счетам и дать возможность переводить деньги.

1) Общая структура интерфейса.

Главная вкладка(maintab):

a) Надпись описывающая содержание вкладки “Мои продукты”
b) Кнопка для создания нового продукта
c) Кнопка для вывода информации о программе
d) Счета пользователя от 0 до 8 штук
e) 3 кнопки для переключения вкладок

Вкладка Информации(paymenttab):

a) Надпись “курсы валют”
b) Кнопка для обновления курса валют
c) Таблица с курсом валют, покупкой и продажей
d) 3 кнопки для переключения вкладок

Вкладка Личного кабинета(accounttab):

a) Надпись “Личный кабинет”
b) надписси с данными пользователя
c) Кнопка для выхода из приложения
d) Кнопка для выхода из аккаунта
e) Карта на которой отмечен оффис банка
f) 3 кнопки для переключения вкладок

Рисунок 3.4.1 – Интерфейс пользователя

2) Логика навигации и сценарии использования.

а) Авторизация пользователя: после ввода логина и пароля открывается главное окно Maintab

b) Выбор вкладки

1)Информация – вкладка на которой находится информация о курсах валют

2)Главная – вкладка на которой можно прссмотреть все свои счета, их баланс и произвести действия с ними(перевод и т.д)

3)Кабинет – вкладка в которой выведена вся информация пользователя(кроме пароля)

c) Просмотр детальной информации: при выборе счета открывается окно в котором показаны все данные о счете, а так же элементы для действия с картой

3.5 Модель данных

Спроектированная модель данных определяет принципы хранения и обмена информацией между различными компонентами приложения. Она решает три ключевые задачи:

- Обеспечение согласованности данных
- Представление информации в пользовательском интерфейсе
- Управление работой СУБД

В реализации используется база данных из двух взаимосвязанных таблиц. Далее приводится их детальная структура с указанием типов полей и существующих связей.

Таблица users – хранит учетные записи пользователей.

Поле	Тип	Описание
id	Integer	Уникальный идентификатор пользователя
telephone	String	телефон (логин) пользователя
password	String	Пароль
Name	String	Имя прользователя
Surname	String	Фамилия пользователя
Passport_number	String	Серия и номер паспорта
Passport_id	String	Индефикационный номер пасспорта
Registered_at	DateTime	Дата и время регистрации
Last_seance	DateTime	Дата и время последнего входа в приложение
Cards	Card	Карты пренадлжежащие пользователю
key	String	Уникальеный ключ пользователя

Таблица card – хранит метаданные о книгах.

Поле	Тип	Описание
id	Integer	Уникальный идентификатор счета
Owner_id	Integer	Id пользователя которому прен. Счет
type	String	Тип счета
Currency	String	Валюта
Balance	Numeric	Баланс
Create_at	DateTime	Дата и время создания счета
Card_number	String	Уникальный номер счета

Owner_card	String	Имя и фамилия владельца
Valid_to	Datetime	Дата до которой счет активен
Cvv	Integer	Cvv код счета(если имеется)
Pin	Integer	Пин код карты
procent	Double	Процентная ставка по карте
limit	Integer	Лимит по карте
Last_transaction	DateTime	Дата и время последней транзакции
Transactions	ARRAY(String)	Список всех транзакций
Last_bank_touch	DateTime	Дата и время последнего расчета процента
owner	user	Владелец карты

Связи между таблицами

1) users ↔ card: 1 пользователь → N счетов;
2) card ↔ user: N счетов → 1 пользователь;

4 Реализация ПО

4.1 Реализация серверной части

Сервер реализован на Python с использованием стандартного модуля socket и библиотеки threading. Основной поток создаёт TCP-сокет, привязывает его к указанному порту и запускает слушатель. Каждое новое подключение обрабатывается в отдельном потоке (handle_client).

Сервер состоит из 3 основных модулей: main.py, service.py, route.py

Задача main.py – обеспечить сетевое взаимодействие. Он создает подключения, и обрабатывает запросы(проверяет на корректность и передает в service.py)

Задача service.py – получает запрос и провод следующие действия в зависимости от метода запроса, например при пост запросе расшифровывает данные, затем передает в route.py, получает ответ, зашифровывает и возвращает

Задача route.py – принимать запрос и выполнять действия с ним в зависимости от комманды которая была задана и возвращает результат действий

Ниже приведена главная функция создания сокета и подключений

```
def main():
    print("Сервер запущен")
    sock = socket.socket()

    print(f"HOST: {HOST}, PORT: {PORT}")
    sock.bind((HOST, PORT))

    sock.listen(50)

    while True:
        conn, addr = sock.accept()

        client_thread = Thread(target=handle_client, args=(conn,))
        client_thread.start()
```

Формат сообщений клиент-серверного взаимодействия - JSON, должен быть размером менее 16384 байт, передается целиком т.к. для работы приложения не требуется передачи большого количества данных, что не требует создания цикла

При подключении нового клиента вызывется функция handle_client которая представляет себя цикл ожидания данных от клиента, и при их получении педеает их в функцию processing_data, который проверяет заголовки запроса и если все верно передает данные в service.py, который смотрит ссылку (команду) и передает соответсвующей функции в route.py

UML-диаграмма классов сервера представлена на рисунке 4.1.1

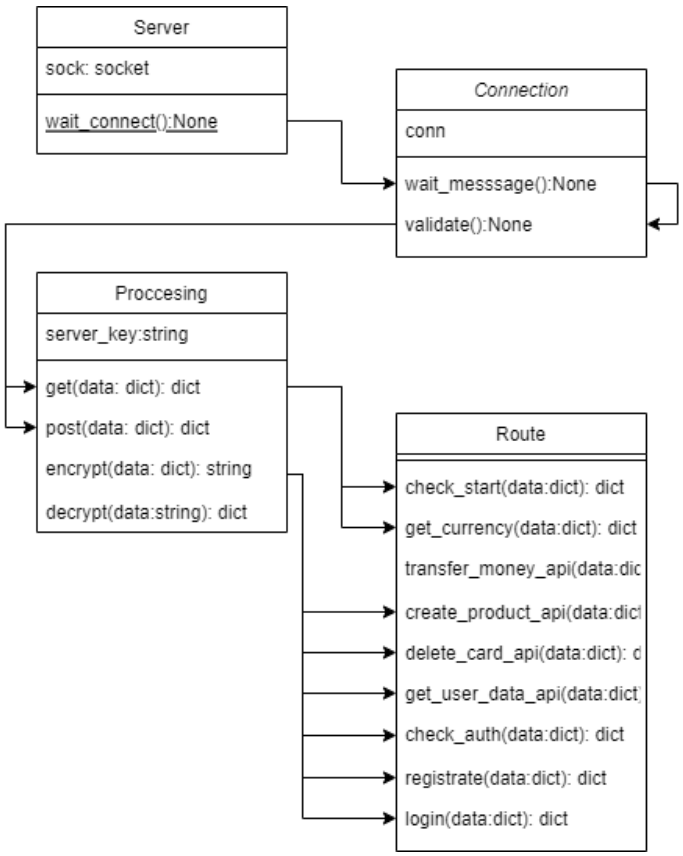


Рисунок 4.1.1 – диаграмма класса сервера

4.2 Реализация клиентской части

Клиентская часть состоит из 3-х основных файов Client.py, service.py и gui_manager.py

Задача Client.py – создавать соединение, правильно строить запросы и передавать данные на сервер

Задача gui_manager.py – строить графичесский интерфейс и визуализировать данные для пользователя, которые приходят с сервера

Задача service.py – связать работу gui_manager с client.py

При входе в приложение gui_manager.py вызывает функцию из service.py которая вызывает метод client.py для отправки данных на сервер, затем, когда сервер дает ответ client.py возвращает данные и service.py проверяет авторизован ли клиент, если сервер прислал новый токен авторизации и данные пользователя то service.py обращается к client.py и обновляет токен, а затем возвращает в gui_manager.py данные о клиенте, он в свою очередь начинает строить интерфейс с данными пользователя

Затем в ходе работы программы происходят аналогичные действия:

gui_manager.py вызывает service.py для валидации и упаковки файлов, service.py вызывает client.py, который добавляет к данным заголовоки(шифрует если выбран метод post), отправляет данные на сервер и возвращает ответ, затем service.py обрабатывает ответ и возвращает готовые данные для gui_manager.py

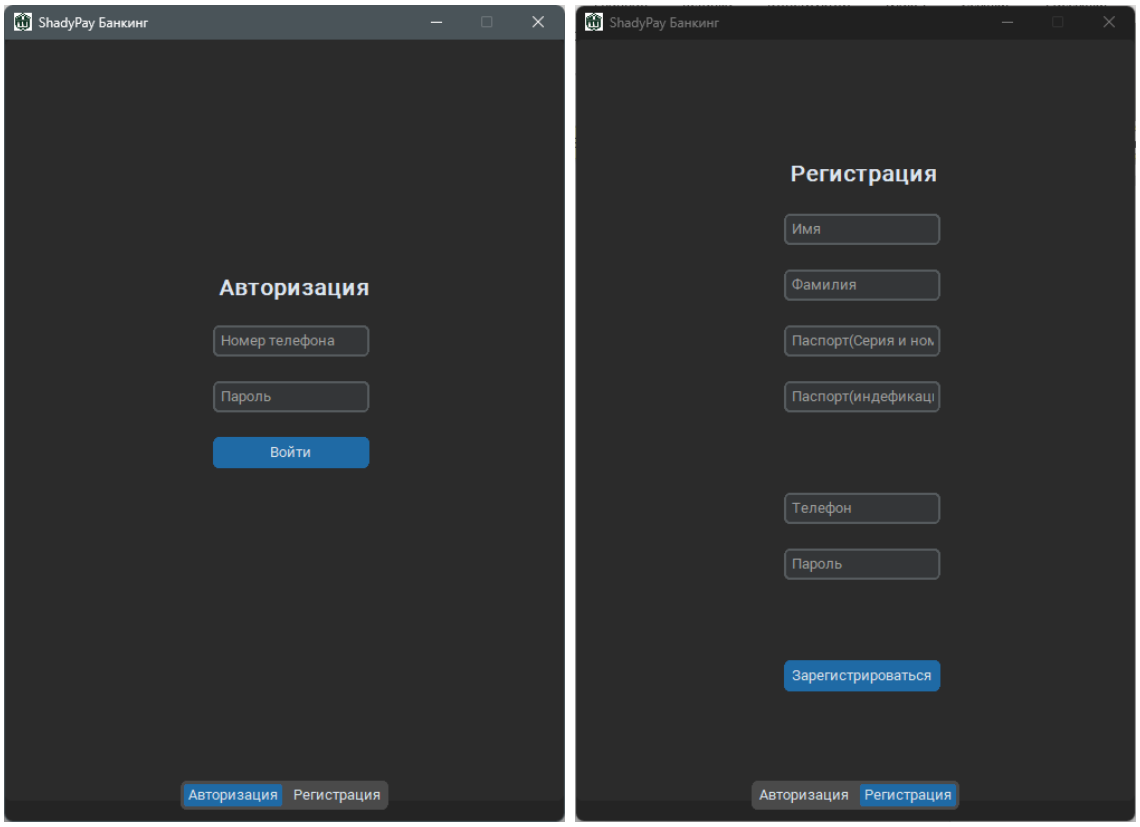
5 Руководство пользователя

Для запуска приложение необходимо перейти в директорию проекта, после чего установить python 3.14, и в коммандной строке написать

>pip inistall -r requirements.txt

Теперь программа доступна для запуска. Чтобы запустить следует запустить main.bat 2

При первом входе откроется окно авторизации/регистрации(рис 5.1) если подключение с сервером успешно, если нет, то изначально откроется окно ввода IP адреса и порта сервера(рис 5.2)



А Б

Рисунок 5.1 - Окно авторизации(А) и регистрации(Б)

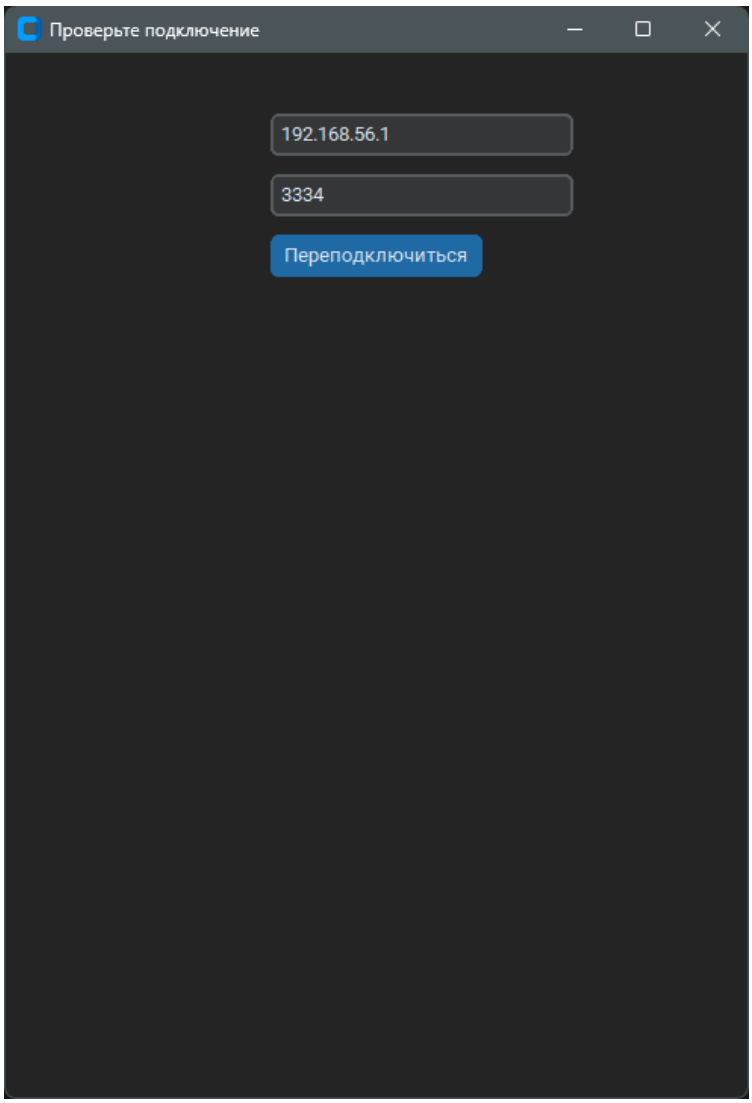


Рисунок 5.2 - Окно ввода IP адреса и порта сервера

При первом входе потребуется зарегистрироваться, для этого нужно в поля ввода ввести свои данные и нажать на кнопку “Зарегистрироваться”

После чего откроется вкладка “Информация”(рис 5.3)

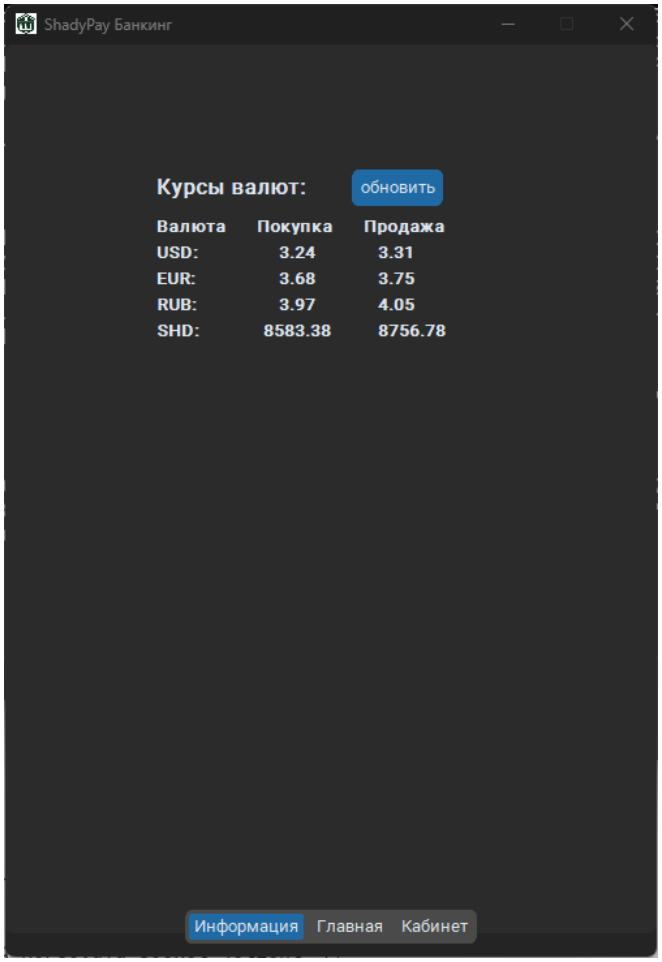


Рисунок 5.3 - Вкладка “Информация”

Теперь нужно нажать снизу на кнопку “Главная”(рис. 5.4), изначально она будет пуста, для добавления счета потребуется на кнопку отмеченную на рисунке 5.4 “Создать новый”, Откроется окно изображенное на рисунке 5.5, в котором нужно выбрать нужные вам данные и нажать на кнопку “Создать”

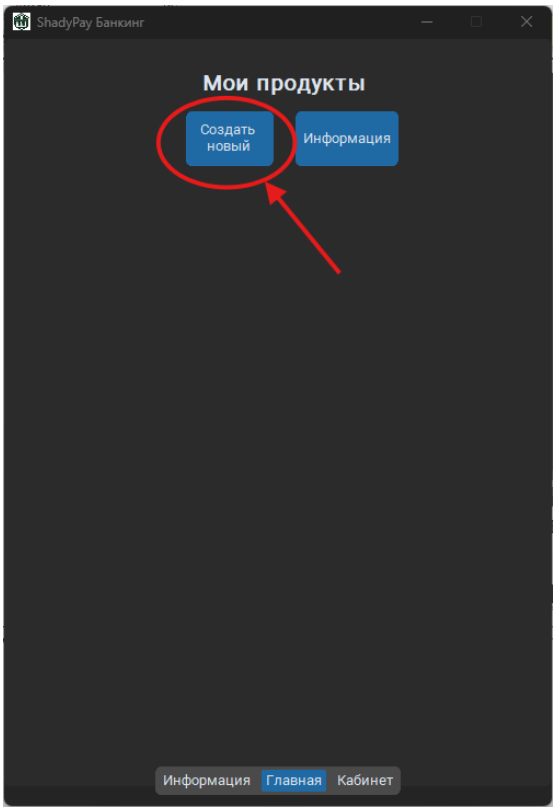


Рисунок - 5.4 Главная вкладка

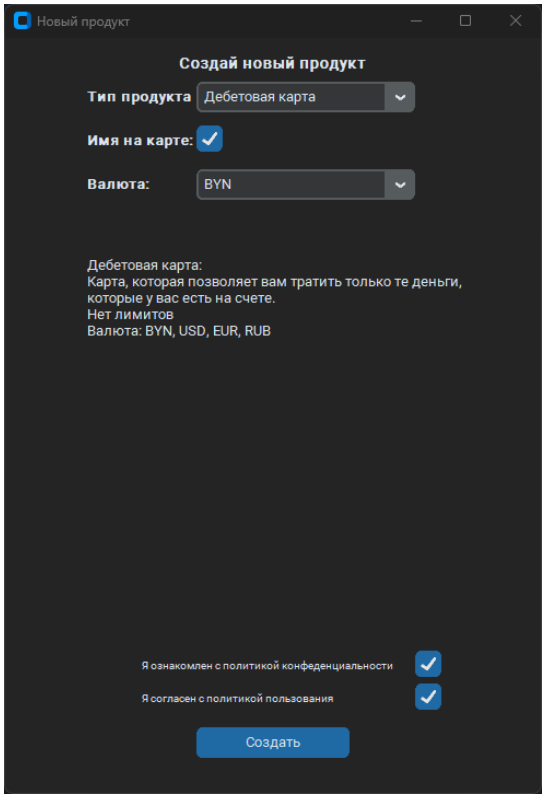


Рисунок 5.5 - Окно создания продукта

Теперь во вкладке главная будет отображаться новый счет, для действий со счетом нужно нажать на кнопку “Открыть” (рисунок 5.6)

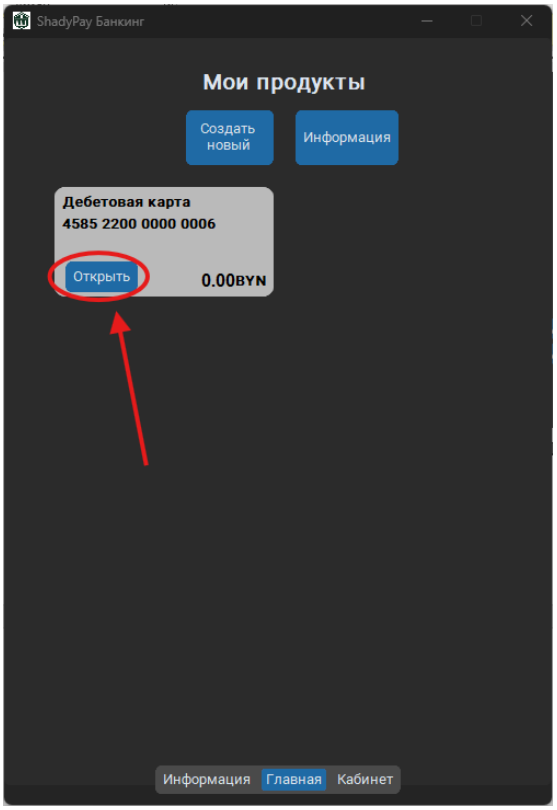


Рисунок 5.6 - главная вкладка с 1 активным счетом

В открывшейся вкладке(рисунок 5.7) можно:

- Перевести деньги по номеру телефона
- Перевести деньги по номеру счета
- Перевести деньги на свой счет
- Удалить карту
- Просмотреть информацию карты нажав на кнопку “Показать”

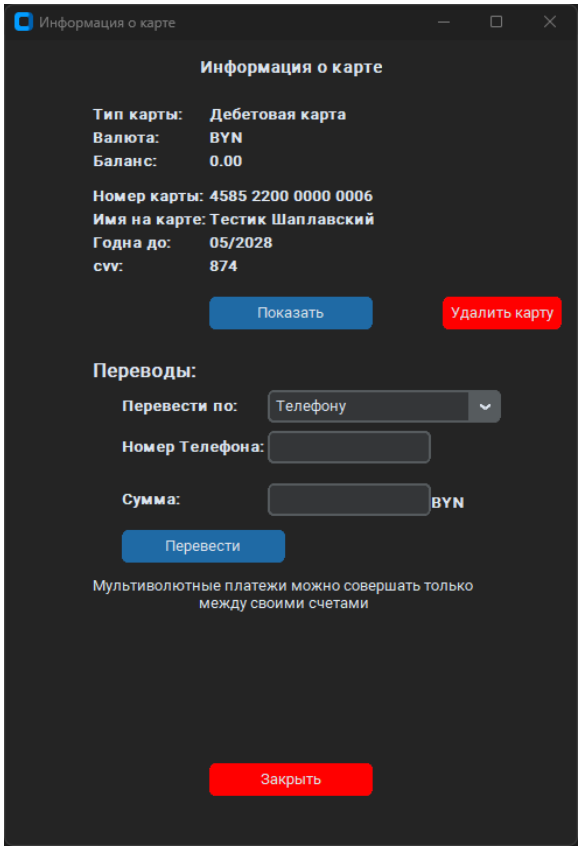


Рисунок 5.7 – Окно управления счетом

Для просмотра информации о программе следует нажать на кнопку сверху (на вкладке “Главная”) (Рисунок 5.4) “Информация”
Для просмотра информации о себе/Выхода из аккаунта/приложения, нужно нажать на кнопку переключения вкладок “Кабинет” и откроется вкладка со всей информацией(Рисунок 5.8)

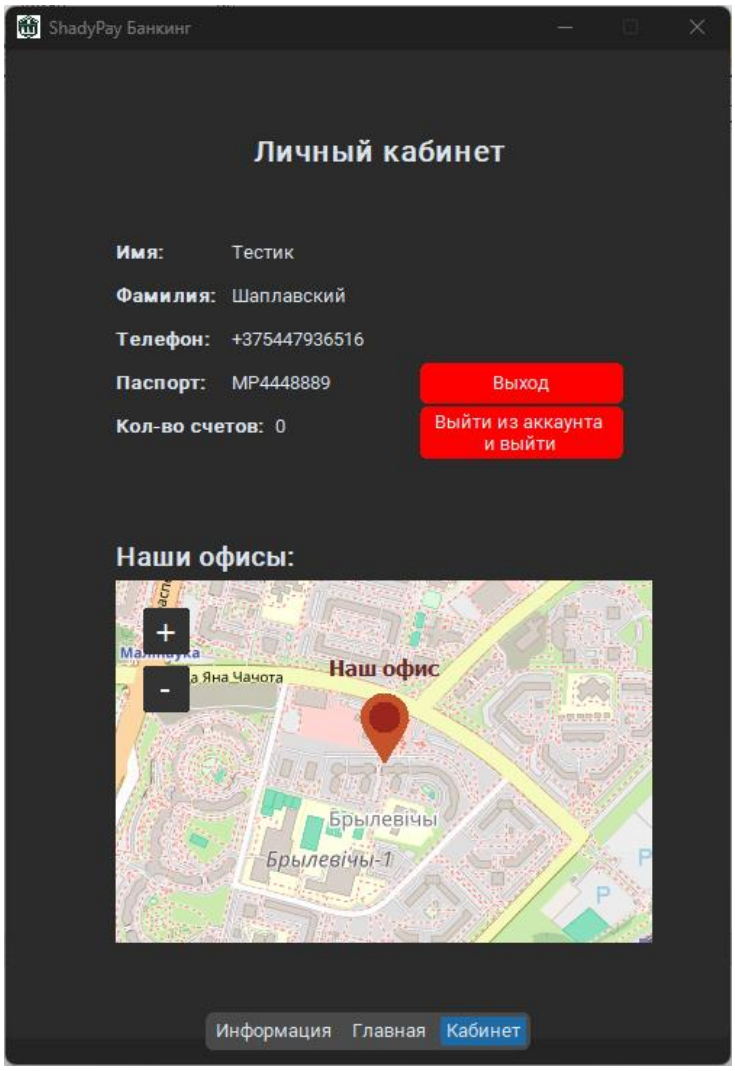


Рисунок 5.8 – Вкладка персонального кабинета

Заключение

В ходе выполнения курсового проекта было разработано клиент-серверное приложение «Банковское приложение» на основе TCP-сокетов, предназначенное для автоматизации базовых банковских операций. Проект реализован на Python с использованием PostgreSQL для хранения данных и обеспечивает безопасное взаимодействие между клиентом и сервером через JSON-форматированные сообщения.

Основные достижения:

1. Архитектура и сетевые технологии:
   - Реализована стабильная клиент-серверная архитектура с многопоточной обработкой запросов.
   - Настроено безопасное соединение с шифрованием данных и аутентификацией через JWT-токены.

2. Функциональность:
   - Разработаны ключевые модули: авторизация, управление счетами, транзакции, курсы валют.
   - Обеспечено разделение ролей (пользователь/администратор) с разграничением прав доступа.

3. Интерфейс:
   - Создан интуитивно понятный GUI на базе CTkinter с тремя основными вкладками: управление счетами, информация о курсах валют и личный кабинет.

4. Оптимизация и безопасность:
   - Данные пользователей хранятся в хешированном виде.
   - Все операции фиксируются в базе данных для аудита.

Приложение соответствует поставленным требованиям, демонстрируя надежность, производительность (время отклика ≤ 2 сек) и удобство использования. Проект может быть расширен за счет интеграции с платежными системами (например, ЕРИП) и добавления мобильной версии.

Список литературы

1. Официальная документация Python (модуль `socket`) - Подробное руководство по работе с сокетами в Python, включая примеры TCP/UDP-серверов.
Ссылка на интернет ресурс: https://docs.python.org/3/library/socket.html

2. PostgreSQL Documentation - Официальная документация по PostgreSQL с примерами SQL-запросов и настройкой БД.
Ссылка на интернет ресурс: https://www.postgresql.org/docs/

3. Real Python: Networking and Sockets Tutorial - Практическое руководство по созданию клиент-серверных приложений на Python.
Ссылка на интернет ресурс: https://realpython.com/python-sockets/

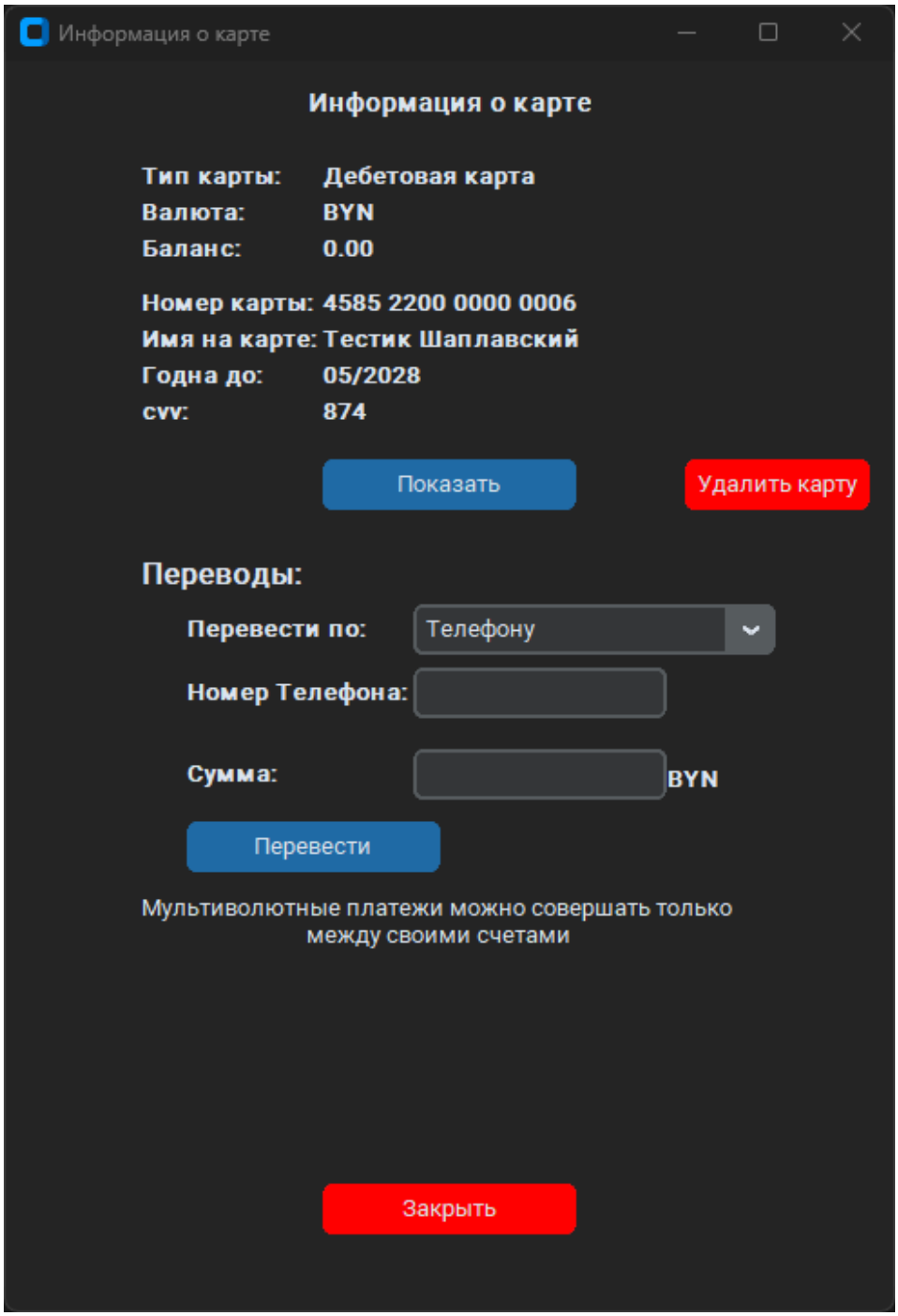
4. MDN Web Docs (JSON и безопасность) - Основы работы с JSON, включая сериализацию и десериализацию данных.
Ссылка на интернет ресурс:
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/JSON

5. GeeksforGeeks: Multithreading in Python - Статьи по многопоточности и параллельному выполнению задач в Python.
Ссылка на интернет ресурс: https://www.geeksforgeeks.org/multithreading-python-set-1/

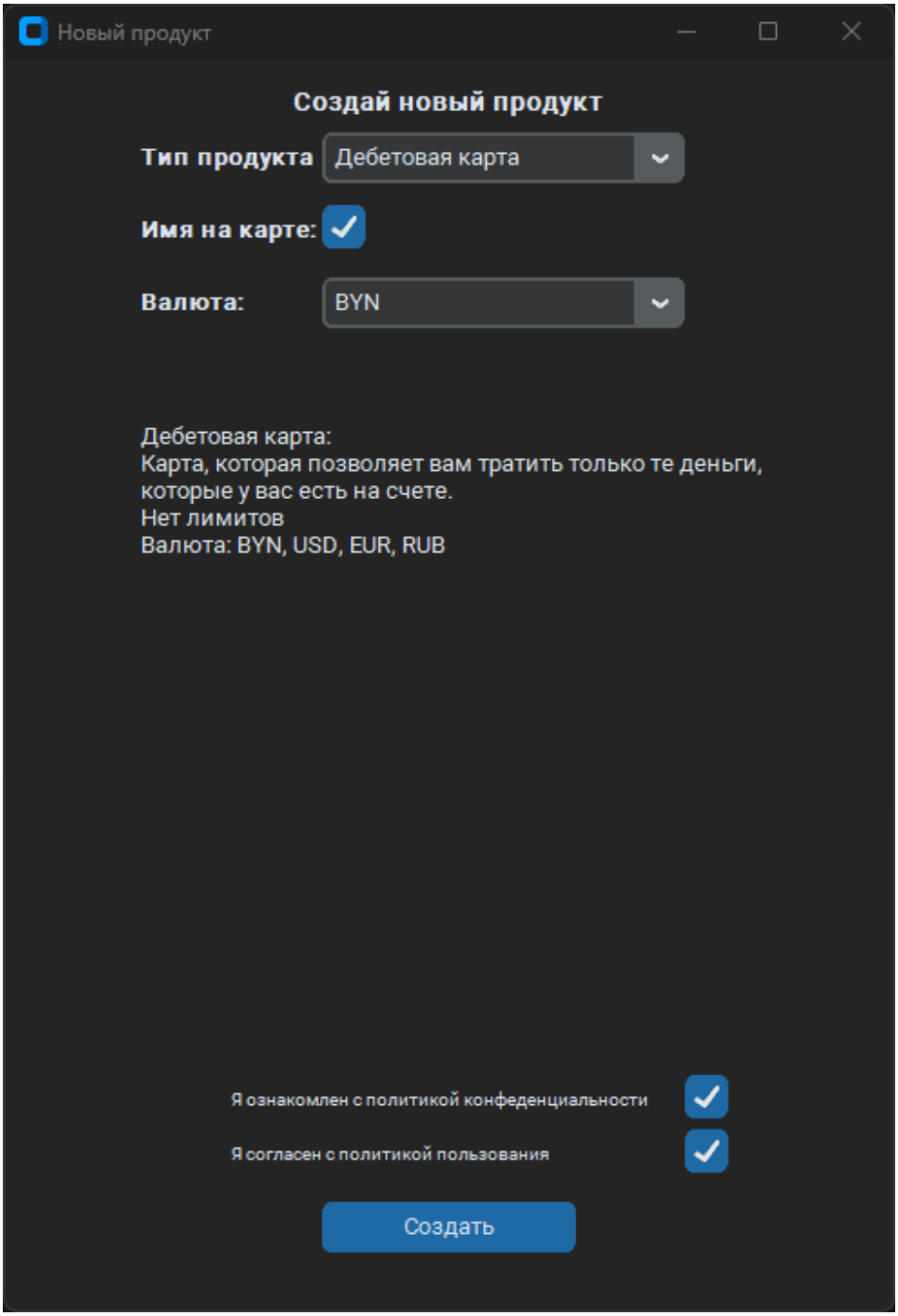
6. JWT.io - Обзор технологии JWT для аутентификации и примеры реализации.
Ссылка на интернет ресурс: https://jwt.io/introduction/

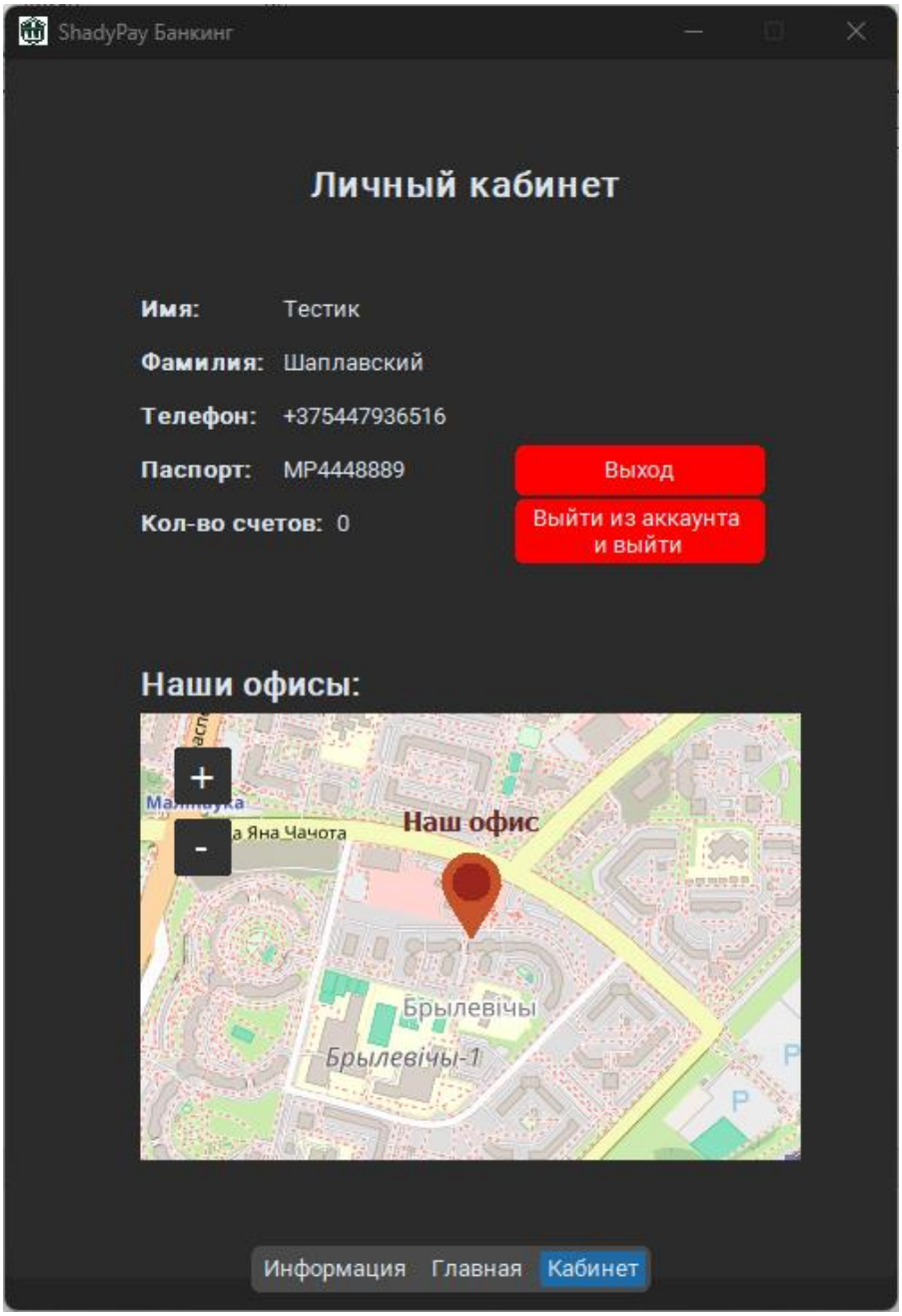
7. Docker Documentation - Руководство по контейнеризации приложений, включая настройку сетей и volumes.
Ссылка на интернет ресурс: https://docs.docker.com/

Приложение А Графическая часть



					КП—1070132317–2025–32			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Шаплавский			Интерфейс пользователя	Лит	Лист	Листов
Руковод.		Белова				У	**2**	**4**
Консульт.		Белова				1–6-05-0612-01 БНТУ, г.Минск		


Новый продукт
Создай новый продукт
Тип продукта
Дебетовая карта
Имя на карте:
Валюта:
BYN
Дебетовая карта:
Карта, которая позволяет вам тратить только те деньги, которые у вас есть на счете.
Нет лимитов
Валюта: BYN, USD, EUR, RUB
Я ознакомлен с политикой конфеденциальности
Я согласен с политикой пользования
Создать



					КП—1070132317–2025–32			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Шаплавский			Интерфейс пользователя	Лит	Лист	Листов
Руковод.		Белова				У	**4**	**4**
Консульт.		Белова				1–6-05-0612-01 БНТУ, г.Минск		

Приложение Б Листинг кода

Серверная часть

Файл main.py

```
#docker save -o shadypay.tar shadypay:latest
import socket
from threading import Thread
from json import dumps, loads
from datetime import datetime

from communicate.service import Proccessing
from config import settings



HOST = settings.host
PORT = int(settings.port)
CONNECTION_TIMEOUT = 600

functions = {
    "get": lambda data: Proccessing.get(data),
    "post": lambda data: Proccessing.post(data),
}

def processing_data(data):
    try:
        data = loads(data.decode())
        print(data)

        if not ('headers' in data or 'data' in data):
            return  dumps({"status": 400, "details": "bad request"}).encode()

        if not data['headers']['config_version'] == settings.config_version:
            return  dumps({"status": 421,  'details': "You need to update app"}).encode()

        try:
            print(f"\nlog: {datetime.now().strftime('%H:%M:%S %d.%m.%Y')}
\nip:{data['headers']['ip']}\n")
        except KeyError:
            return  dumps({'status': 403, 'details': 'Forbidden. Need you ip'}).encode()

        try:
            answer = functions[data['headers']['method']](data)
        except KeyError as e:
            raise e  # TODO: для отладки
            return dumps( {"status": 404, 'details': 'Method not found'}).encode()
        except Exception as e:
            raise e  # TODO: для отладки
            print(f"Error processing request: {e}")
            return dumps( {"status": 501, "details": f"Internal Server Error: {str(e)}"}).encode()
    except Exception as e:
        print(f"log {datetime.now()}: {e}")
        raise e #TODO: для отладки
        return dumps({"status": 400, "details": "Bad request"}).encode()


    if not 'status' in answer.keys():
        answer['status'] = 200

    answer = dumps(answer).encode()
    return answer



def handle_client(conn):
    conn.settimeout(CONNECTION_TIMEOUT)
    while True:
        try:
```

```
            data = conn.recv(16384)

            if not data:
                continue

            conn.settimeout(CONNECTION_TIMEOUT)
            answer = processing_data(data)
            print("23432",answer)
            conn.send(answer)

        except socket.timeout:
            print(f"log {datetime.now()}: тайм-аут соединения")
            break

        except ConnectionResetError:
            print(f"log {datetime.now()}: подключение неожиданно разорвано")
            conn.close()
            break
        except OSError:
            print(f"log {datetime.now()}: соединение закрыто")
            conn.close()
            break

        except Exception as e:
            print(e)
            answer  = {"status": 500, "details": str(e)}
            conn.send(dumps(answer).encode())
            break

    conn.close()


def main():
    print("Сервер запущен")
    sock = socket.socket()

    print(f"HOST: {HOST}, PORT: {PORT}")
    sock.bind((HOST, PORT))

    sock.listen(50)

    while True:
        conn, addr = sock.accept()

        client_thread = Thread(target=handle_client, args=(conn,))
        client_thread.start()

if __name__ == "__main__":
    main()
```

Файл service.py

```
from json import loads, dumps
from cryptography.fernet import Fernet
import base64

from communicate.route import router_dir
from config import settings



class Proccessing:
    server_key = settings.secret_server_key
    @staticmethod
    def post(data) -> dict:
        try:
            routers = router_dir['post']
            if data['headers']['route'] not in routers:
                return {"status": 404, "details": "Route not found"}

            # Расшифровываем данные
            try:
```

```
            data['data'] = Proccessing.decryption(Proccessing.server_key, data['data'])
            print(data) # TODO: для отладки
        except Exception as e:
            print(f"Error decrypting data: {e}")
            return {"status": 400, "details": "Failed to decrypt data"}


        answer = routers[data['headers']['route']](data)

        encrypted_data = Proccessing.encryption(Proccessing.server_key, answer)
        responce = {'data': encrypted_data}

        try:
            if 'details' and 'status' in answer.keys():
                responce["status"] = answer['status']
                responce['details'] = answer['details']
        except AttributeError as e:
            responce = {"status": 200, "details": "no answer"}

        return responce


    except Exception as e:
        raise e #TODO: для отладки
        print(f"Unexpected error in post: {e}")
        return {"status": 500, "details": f"Internal server error: {str(e)}"}

@staticmethod
def get(data) -> dict:
    try:
        routers = router_dir['get']
        print(data) # TODO: для отладки
        if data['headers']['route'] not in routers:
            return {"status": 404, "details": "Route not found"}

        answer = {}
        answer.update(routers[data['headers']['route']](data))

        responce = {'data': answer}

        try:
            if 'details' and 'status' in answer.keys():
                responce["status"] = answer['status']
                responce['details'] = answer['details']
        except AttributeError:
            responce = {"status": 200, "details": "no answer"}

        return responce
    except Exception as e:
        print(f"Error in get: {e}")
        raise e # TODO: для отладки
        return {"status": 500, "details": f"Internal server error: {str(e)}"}

@staticmethod
def encryption(key, data):
    try:
        cipher_suite = Fernet(key)
        json_data = dumps(data).encode()
        encrypted_data = cipher_suite.encrypt(json_data)
        return base64.b64encode(encrypted_data).decode()
    except Exception as e:
        print(f"Error in encryption: {e}")
        raise

@staticmethod
def decryption(key, data):
    try:
        cipher_suite = Fernet(key)
        encrypted_data = base64.b64decode(data)
        decrypted_data = cipher_suite.decrypt(encrypted_data)
        print(f"ЗАШИФРОВАННО: {data}") # TODO: для отладки
        data = loads(decrypted_data.decode())
        print(f"РАСШИФРОВАННО: {data}") # TODO: для отладки
        return data
```

```
        except Exception as e:
            print(f"Ошибка при расшифровке: {str(e)}")
            raise
```

Файл route.py

```
from datetime import datetime

from user.dao import UsersDAO
from user.auth import get_password_hash, authenticate_user, create_access_token, get_current_user
from user.service import get_user_data
from card.service import *
from background_process.currency import currency



router_dir = {
    'get': {},
    'post': {},
}

def router(method, route):
    def decorator(func):
        router_dir[method][route] = func
        return func
    return decorator


@router('get', 'check_start')
def check_start(data = None):
    return {"status": 200, "details": "Сервер запущен, get: get_congif для подробной информации"}

@router('get', 'get_config')
def get_name(data = None):
    return {}

@router('get', 'currency_api')
def get_currency(data = None):
    return currency.to_dict()

@router('post', 'transfer_money_api')
def transfer_money_api(data):
    user = get_current_user(data)
    if not user:
        return {"status": 401, "details": "Unauthorized"}
    answer =transfer_money(user, data['data'])
    return answer

@router('post', 'get_balance')
def get_balance(data):
    print(data)

@router('post', 'create_product_api')
def create_product_api(data):
    user = get_current_user(data)
    if not user:
        return {"status": 401, "details": "Unauthorized"}
    try:
        add_product(user, data['data'])
    except Exception as e:
        return {"status": 400, "details": f"{str(e)}"}

@router('post', 'delete_card_api')
def delete_card_api(data):
    user = get_current_user(data)
    if not user:
        return {"status": 401, "details": "Unauthorized"}

    return delete_card(user, data['data'])


@router('post', 'get_user_data_api')
def get_user_data_api(data):
```

```
    user = get_current_user(data)
    if not user:
        return {"status": 401, "details": "Unauthorized"}
    user_data = get_user_data(user)
    print(user_data)
    return user_data


@router('post', 'check_auth')
def check_auth(data):
    user = get_current_user(data)

    if not user:
        return {"status": 401, "details": "Unauthorized"}

    UsersDAO.update_one(user.id, last_seance=datetime.now())
    jwt = create_access_token(data={"sub": str(user.id)})
    return {"JWT": jwt, "Auth": True}

@router('post', 'registration')
def registrate(data):
    data = data['data']
    try:
        user = UsersDAO.add_user(
            name=data['name'],
            surname=data['surname'],
            passport_number=data['passport_number'],
            passport_id=data['passport_id'],
            telephone=data['telephone'],
            password=get_password_hash(data['password'])
        )

        # Получаем данные пользователя до закрытия сессии
        user_id = user.id
        user_key = user.key

        jwt = create_access_token(data={"sub": str(user_id)})
        return {'JWT': jwt, "key": user_key}
    except Exception as e:
        return {"status": 500, "details": f"Registration error: {str(e)}"}


@router('post', 'login')
def login(data):
    print(data)
    try:
        data = data['data']
        user = authenticate_user(data['telephone'], data['password'])
    except KeyError:
        return {"status": 401, "details": "Bad data"}
    except AttributeError:
        return {"status": 403, "details": "Неверный логин или пароль"}

    if not user:
        return {"status": 403, "details": "Неверный логин или пароль"}
    jwt = create_access_token(data={"sub": str(user.id)})
    key = user.key
    return {'JWT': jwt, "key":key}


print(router_dir['get']['check_start'](None))
```

Клиентская часть

Файл client.py

```
import socket
import base64
```

```
from cryptography.fernet import Fernet
from json import loads, dumps, load, dump
from time import sleep


class Client:
    def __init__(self):
        self.server_key =b"3nBGTLyXjpz_X-CLFtkEVnm6TdwoX2Igm_3wll1JLek="

        self.config = {}
        with open("data/server_config.json", "r") as json_file:
            self.config = load(json_file)
        if not self.config["ip"]:
            self.config["ip"] = socket.gethostbyname(socket.gethostname())
            with open("data/server_config.json", "w") as json_file:
                dump(self.config, json_file)


        self.sock = socket.socket()
        try:
            self.sock.connect((self.config["host"], self.config["port"]))
        except Exception as e:
            pass

        self.header_pattern = {
            'method': '',
            'route': '',
            'JWT': self.config["JWT"],
            "ip": self.config["ip"],
            "config_version": self.config["config_version"]
        }


    def write_config(self):
        with open("data/server_config.json", "w") as json_file:
            dump(self.config, json_file)


    def update_json(self):
        with open("data/server_config.json", "r") as json_file:
            self.config = load(json_file)


    def update_jwt(self, jwt):
        self.config["JWT"] = jwt
        with open("data/server_config.json", "w") as json_file:
            dump(self.config, json_file)

        self.header_pattern['JWT'] = jwt


    def reconnect(self):
        try:
            self.sock.close()
            self.sock = socket.socket()
            self.sock.connect((self.config["host"], self.config["port"]))
            sleep(0.5)
            return True
        except Exception as e:
             raise e
             return False


    def change_connection(self, ip, port):
        self.config["host"] = ip
        self.config["port"] = port
        self.write_config()


    def get(self, route, data: dict = {"details": "No data"}) -> dict:
        try:
            headers = self.header_pattern.copy()
            headers['method'] = 'get'
```

```
            headers['route'] = route
            request = {'headers': headers, 'data': data}
            self.sock.send(dumps(request).encode())
        except OSError as e:
            if self.reconnect():
                return  self.get(route, data)
            else:
                raise OSError("нет подключения, Попробуйте позже")

        answer = self.sock.recv(16384)
        answer = loads(answer.decode())
        print(f"\n\nКлиентом получено(зашифрованно): {answer}")
        if not 200 <= answer["status"] <= 399:
            details = ""
            if answer['details']:
                details = answer['details']
            raise ConnectionError(details)

        self.check_answer(answer)
        return answer


    def post(self, route, data: dict = {"details": "No data"}) -> None:
        try:
            encrypt_data = self.encryption(self.server_key, data)

            headers = self.header_pattern.copy()
            headers['method'] = 'post'
            headers['route'] = route

            request = {'headers': headers, 'data': encrypt_data}

            self.sock.send(dumps(request).encode())
        except OSError as e:
            if self.reconnect():
                return self.post(route, data)
            else:
                raise OSError("нет подключения")

        answer = self.sock.recv(16384)
        answer = loads(answer.decode())

        if not 200 <= answer["status"] <= 399:
            details = ""
            if answer['details']:
                details = answer['details']
            raise ConnectionError(details)

        print(f"\n\nКлиентом получено(зашифрованно): {answer}")
        try:
            answer['data'] = self.decryption(self.server_key, answer['data'])
            print(f"\nКлиентом получено(расшифрованно): {answer}")
        except KeyError:
            answer['data'] = None

        return answer


    def check_answer(self, answer):
        if not 200 <= answer["status"] <= 299:
            details = ""
            if answer['details']:
                details = answer['details']
            raise ConnectionError(details)


    def encryption(self, key, data):
        cipher_suite = Fernet(key)
        json_data = dumps(data).encode()
        encrypted_data = cipher_suite.encrypt(json_data)

        return base64.b64encode(encrypted_data).decode()
```

```
    def decryption(self, key, data):
        cipher_suite = Fernet(key)

        encrypted_data = base64.b64decode(data)
        decrypted_data = cipher_suite.decrypt(encrypted_data)

        data =loads(decrypted_data.decode())

        return data


    def close_connection(self):
        self.sock.close()


client = Client()
```

Файл service.py

```
from communicate.client import client
from json import load, dump
import asyncio



def delete_card_serv(card_number)-> dict:
    """Удаляет карту по номеру карты"""
    data = {
        'card_number': card_number
    }
    answer = client.post('delete_card_api', data)
    try:
        return answer['details']
    except KeyError:
        raise ConnectionError('Ошибка удаления карты')


def get_currency() -> dict:
    """Получает курс валют"""
    answer = client.get('currency_api')
    try:
        currency = answer["data"]

        currency["RUB"]['buy'] *= 0.01
        currency['RUB']['sell'] *= 0.01
        for i in currency:
            currency[i]['buy'] = round(1/currency[i]['buy'], 2)
            currency[i]['sell'] = round(1/currency[i]['sell'], 2)

        return currency

    except KeyError:
        return answer['details']


def transfer_service(card_number, adr, sum, transfer_type) -> dict:
    """
    Переводит деньги с карты на карту
    args: card_number - номер карты, adr - номер карты или телефон, sum - сумма перевода, transfer_type
- тип перевода
    """
    data = {
        'card_number': card_number,
        "transfer_type": transfer_type,
        'adr': adr,
        'sum': sum
    }
    answer = client.post('transfer_money_api', data)
    return answer["data"]['details']


def create_product(product_type, is_named_product, currency)-> None:
    """ Создает карту/счет """
```

```
    data = {
        'product_type': product_type,
        'is_named_product': is_named_product,
        'currency': currency
    }
    try:
        client.post('create_product_api', data)
    except ConnectionError as e:
        raise e


def quit_account() -> None:
    """Выход из аккаунта"""
    with open("data/server_config.json", "r") as json_file:
        config = load(json_file)

    config['JWT'] = None
    config["key"] = None
    client.config['JWT'] = None
    client.config["key"] = None

    with open("data/server_config.json", "w") as json_file:
        dump(config, json_file)

    client.update_json()

def check_auth() -> bool:
    """Проверяет авторизацию"""
    try:
        answer = client.post('check_auth', {})
    except ConnectionRefusedError as e:
        raise e
    except ConnectionError:
        return False
    except Exception as e:
       raise ConnectionRefusedError

    try:
         client.update_jwt(answer['data']['JWT'])
    except KeyError:
        return False
    return True


def get_user_data()-> dict:
    """Получает данные пользователя"""
    answer = client.post("get_user_data_api", {})
    try:
        user_data = answer['data']
    except KeyError:
        return answer['details']
    return user_data

async def login(phone, password)->bool:
    """Авторизация"""
    if not phone or not password:
        raise ValueError('Все поля должны быть заполнены')

    answer = client.post('login', {
        'telephone': phone,
        'password': password
    })

    with open("data/server_config.json", "r") as json_file:
       config = load(json_file)

    try:
        jwt_token = answer['data']['JWT']
        config['JWT'] = jwt_token
        config["key"] = answer['data']["key"]
        # Обновляем JWT в клиенте
        client.update_jwt(jwt_token)
    except KeyError:
        raise ConnectionAbortedError('Неверный логин или пароль')
```

```
    with open("data/server_config.json", "w") as json_file:
        dump(config, json_file)

    client.update_json()

    return True


async def registration(name, surname, passport_number, passport, phone, password)-> bool:
    """Регистрация"""
    if not name or not surname or not passport_number or not passport or not phone or not password:
        raise ValueError('Все поля должны быть заполнены')

    if len(name) < 3 or len(surname) < 3:
        raise ValueError('Имя и фамилия должны быть длиннее 3 символов')

    if len(passport) != 14:
        raise ValueError('ID паспорта должен быть 14 символов')
    if len(passport_number) != 9:
        raise ValueError('Номер паспорта должен быть 9 символов')

    phone = phone.replace(' ', '')
    phone = phone.replace('-', '')
    phone = phone.replace('(', '')
    phone = phone.replace(')', '')
    if len(phone) != 13:
        raise ValueError('Номер телефона должен быть 13 символов')

    answer = client.post('registration', {
        'name': name,
        'surname': surname,
        'passport_number': passport_number,
        'passport_id': passport,
        'telephone': phone,
        'password': password
    })

    with open("data/server_config.json", "r") as json_file:
       config = load(json_file)

    jwt_token = answer['data']['JWT']
    config['JWT'] = jwt_token
    config["key"] = answer['data']["key"]
    # Обновляем JWT в клиенте
    client.update_jwt(jwt_token)

    with open("data/server_config.json", "w") as json_file:
        dump(config, json_file)

    client.update_json()

    return True
```