

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Модели данных и системы управления базами данных

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему

ВЕБ-ПРИЛОЖЕНИЕ ДЛЯ РАБОТЫ ЗООМАГАЗИНА

БГУИР КП 1-40 04 01 029 ПЗ

Студент

Д. С. Шевцова

Руководитель

А. В. Давыдчик

Минск 2024

СОДЕРЖАНИЕ

Введение.....	4
1 Архитектура вычислительной системы.....	5
1.1 Структура и архитектура вычислительной системы.....	5
1.2 История, версии и достоинства.....	7
1.3 Обоснование выбора вычислительной системы.....	9
2 Платформа программного обеспечения.....	11
2.1 Выбор операционной системы.....	11
2.2 Выбор платформы для написания программы.....	12
3 Теоретическое обоснование разработки программного продукта.....	13
3.1 Обоснование необходимости разработки.....	13
3.2 Технологии программирования, используемые для решения поставленных задач.....	13
4 Проектирование функциональных возможностей программы.....	14
4.1 Подключение к базе данных.....	14
4.2 Регистрация и авторизация пользователей.....	14
4.3 Управление пользователями.....	14
4.4 Взаимодействие с сущностями приложения.....	14
4.6 Руководство пользователя.....	14
5 Проектирование разрабатываемой базы данных программного обеспечения.....	15
5.1 Разработка информационной модели.....	15
5.2 ER-диаграмма базы данных.....	15
5.3 Оптимизация и целостность разработанной базы данных.....	15
5.4 Описание базы данных.....	15
Заключение.....	16
Список литературных источников.....	17
Приложение А (обязательное) Листинг программного кода.....	18
Приложение Б (обязательное) Конечная схема базы данных.....	19
Приложение В (обязательное) Ведомость курсового проекта.....	20

ВВЕДЕНИЕ

В современном мире бизнес сталкивается с необходимостью оперативно реагировать на вызовы, связанные с развитием технологий и изменениями потребностей клиентов. Наряду с традиционными магазинами для людей, активно развиваются зоомагазины, ориентированные на владельцев домашних животных. Этот рынок динамично растет, что приводит к высокой конкуренции. В условиях постоянных изменений бизнес должен обеспечить доступность информации и удовлетворить требования клиентов. Всё больше потребителей предпочитают онлайн-шоппинг и избегают походов в физические магазины.

Целью данной курсовой работы является создание веб-приложения для зоомагазина, которое будет включать базу данных для хранения и управления информацией о товарах, клиентах, заказах, питомцах и услугах. Разработанное приложение должно упростить процесс администрирования и ускорить доступ к информации, что особенно важно для обеспечения качественного обслуживания клиентов.

Исходя из цели проекта был составлен следующий перечень задач:

- 1 Определить и обосновать перечень информационных сущностей для выбранной предметной области.
- 2 Разработать структуру базы данных, которая охватывает все необходимые аспекты работы зоомагазина.
- 3 Реализовать механизм взаимодействия с сущностями приложения.
- 4 Создать приложение, использующее разработанную базу данных.
- 5 Создать графический интерфейс для взаимодействия с приложением, использующим разработанную базу данных.

В ходе разработки программного средства будут использованы принципы проектирования и современные технологии для создания надежной и безопасной базы данных, которая станет важным инструментом для работы зоомагазина. В конечном итоге данная система должна упростить рабочие процессы и минимизировать ошибки, связанные с человеческим фактором, что будет способствовать повышению качества обслуживания посетителей.

1 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

1.1 Структура и архитектура вычислительной системы

PostgreSQL – это объектно-реляционная система управления базами данных (ORDBMS), наиболее развитая из открытых СУБД в мире. Имеет открытый исходный код и является альтернативой коммерческим базам данных. С помощью PostgreSQL можно создавать, хранить базы данных и работать с данными с помощью запросов на языке SQL.

Одной из наиболее сильных сторон СУБД PostgreSQL является архитектура. Как и в случаях со многими коммерческими СУБД, PostgreSQL можно применять в среде клиент-сервер – это предоставляет множество преимуществ и пользователям, и разработчикам.

В основе PostgreSQL – серверный процесс базы данных, выполняемый на одном сервере. Доступ из приложений к данным базы PostgreSQL производится с помощью специального процесса базы данных. То есть клиентские программы не могут получать самостоятельный доступ к данным даже в том случае, если они функционируют на том же ПК, на котором осуществляется серверный процесс. пользователям, и разработчикам.

В основе PostgreSQL – серверный процесс базы данных, выполняемый на одном сервере. Доступ из приложений к данным базы PostgreSQL производится с помощью специального процесса базы данных. То есть клиентские программы не могут получать самостоятельный доступ к данным даже в том случае, если они функционируют на том же ПК, на котором осуществляется серверный процесс.

Типичная модель распределенного приложения СУБД PostgreSQL (рисунок 1.1):

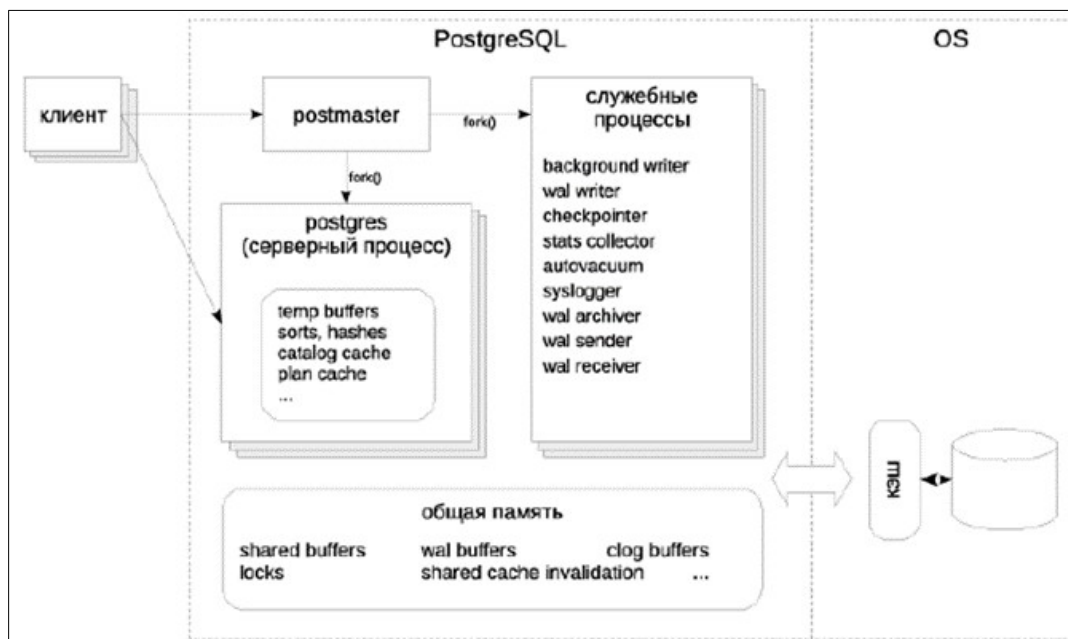


Рисунок 1.1 – Схема СУБД PostgreSQL

СУБД PostgreSQL ориентирована на протокол TCP/IP (локальная сеть либо Интернет), при этом каждый клиент соединён с главным серверным процессом БД (на рисунке 1.1 этот процесс обозначен Postmaster). Именно Postmaster создает новый серверный процесс специально в целях обслуживания запросов на доступ к данным определенного клиента.

Сервер PostgreSQL может обрабатывать несколько одновременных подключений от клиентов. Для этого он запускает новый процесс для каждого соединения. С этого момента клиент и новый серверный процесс обмениваются данными без вмешательства исходного процесса postgres. Таким образом, процесс сервера-супервизора всегда работает, ожидая клиентских подключений, в то время как клиентские и связанные серверные процессы приходят и уходят.

Данные, которыми управляет PostgreSQL, хранятся в базах данных. Один экземпляр PostgreSQL одновременно работает с несколькими базами, которые вместе называются кластером баз данных.

Каталог, в котором размещаются все файлы, относящиеся к кластеру, обычно называют словом PGDATA, по имени переменной окружения, указывающей на этот каталог.

При инициализации в PGDATA создаются три одинаковые базы данных (рисунок 1.2):

1 template0 используется, например, для восстановления из логической резервной копии или для создания базы в другой кодировке и никогда не должна меняться;

2 template1 служит шаблоном для всех остальных баз данных, которые может создать пользователь в этом кластере;

3 postgres представляет собой обычную базу данных, которую можно использовать по своему усмотрению.

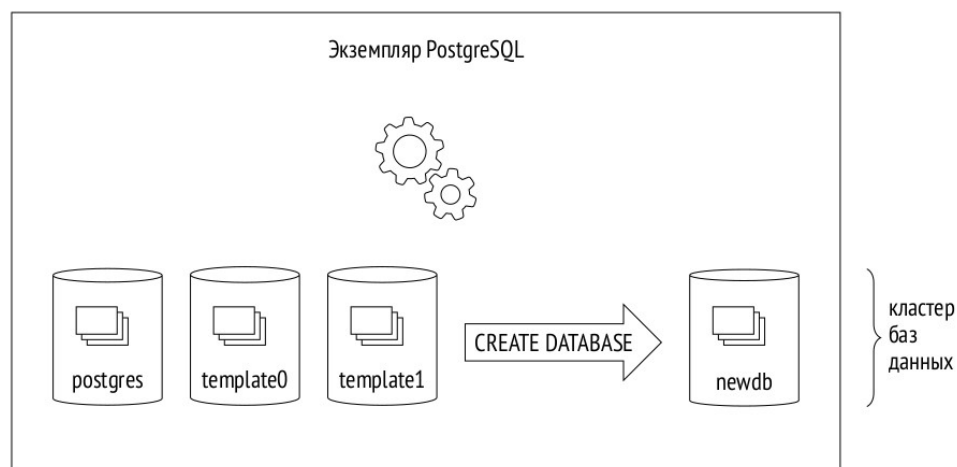


Рисунок 1.2 – Кластер PostgreSQL

Метаинформация обо всех объектах кластера (таких как таблицы, индексы, типы данных или функции) хранится в таблицах, относящихся к системному каталогу. В каждой базе данных имеется собственный набор таблиц (и представлений), описывающих объекты этой конкретной базы. Существует также несколько таблиц системного каталога, общих для всего кластера, которые не принадлежат какой-либо определенной базе данных и доступны в любой из них.

1.2 История, версии и достоинства

Ранние версии системы были основаны на старой программе POSTGRES University, созданной университетом Беркли: так появилось название PostgreSQL. И сейчас СУБД иногда называют «Постгрес». Существуют сокращения PSQL и PgSQL – они тоже обозначают PostgreSQL.

По состоянию на июнь 2024 года PostgreSQL занимает четвертое место в общемировом рейтинге популярных СУБД (рисунок 1.1).

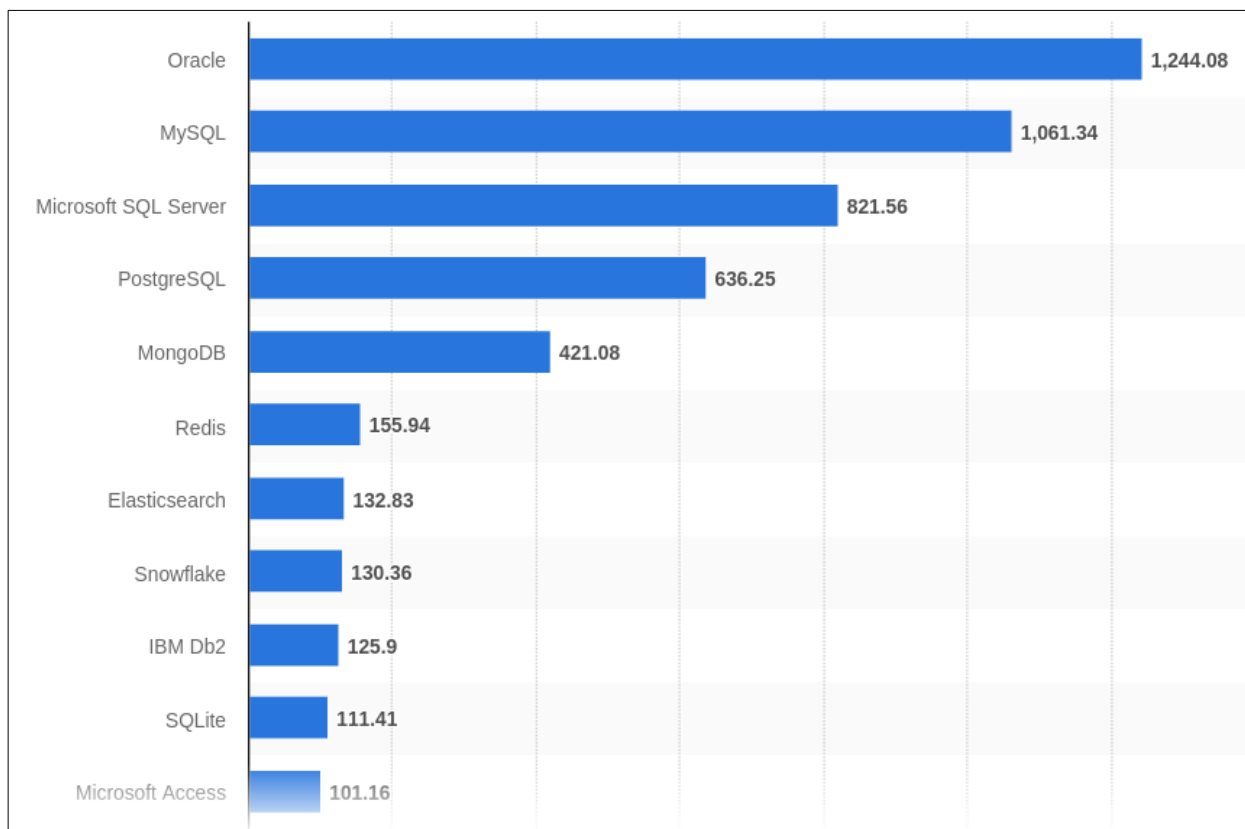


Рисунок 1.1 – Рейтинг популярности СУБД в июне 2024 года

У СУБД PostgreSQL много преимуществ, которые продолжают повышать ее популярность:

1 Любой специалист может бесплатно скачать, установить СУБД и сразу начать работу с базами данных.

2 PostgreSQL подходит для работы в любой операционной системе: Linux, macOS, Windows. Пользователь получает систему «из коробки» – чтобы установить и использовать программу, не нужны дополнительные инструменты.

3 PostgreSQL поддерживает много разных типов и структур данных, в том числе сетевые адреса, данные в текстовом формате JSON и геометрические данные для координат геопозиций. Все эти форматы можно хранить и обрабатывать в СУБД. Также при работе с PostgreSQL можно создавать собственные типы данных, их называют пользовательскими.

4 Размер базы данных в PostgreSQL не ограничен и зависит от того, сколько свободной памяти есть в месте хранения: на сервере, локальном компьютере или в облаке.

5 PostgreSQL реализует принципы ACID. Это четыре требования для надежной работы систем, которые обрабатывают данные в режиме реального

времени. Если все требования выполняются, данные не будут теряться из-за технических ошибок или сбоев в работе оборудования.

6 PostgreSQL поддерживает все современные функции баз данных: оконные функции, вложенные транзакции, триггеры.

7 Хотя большинство операций в PostgreSQL и используют классический стандарт языка SQL, помимо него поддерживается и свой отдельный диалект, позволяющий еще комфортнее писать запросы.

8 Поддерживается репликация «из коробки». Репликация – это сохранение копии базы данных. Копия может находиться на другом сервере.

9 PostgreSQL позволяет быстро без потерь перенести данные из другой СУБД.

10 Возможность одновременного доступа к базе с нескольких устройств. В СУБД реализована клиент-серверная архитектура, когда база данных хранится на сервере, а доступ к ней осуществляется с клиентских компьютеров. Для ситуаций, когда несколько человек одновременно модифицируют базу используется технология MVCC – Multiversion Concurrency Control, многоверсионное управление параллельным доступом.

Благодаря перечисленным выше преимуществам иногда PostgreSQL называют бесплатным аналогом Oracle Database. Обе системы адаптированы под большие проекты и высокую нагрузку. Но есть разница: они по-разному хранят данные, предоставляют разные инструменты и различаются возможностями. Важная особенность PostgreSQL в том, что эта система – feature-rich: так называют проекты с широким функционалом.

1.3 Обоснование выбора вычислительной системы

Для разработки приложения для зоомагазина была выбрана система управления базами данных PostgreSQL, так как она сочетает в себе высокую функциональность, надежность и популярность среди реляционных СУБД. К июню 2024 года PostgreSQL занимает одно из ведущих мест среди мировых СУБД, что подтверждает ее востребованность в разных областях, включая торговлю и онлайн-бизнес.

Одним из главных преимуществ PostgreSQL является свободная лицензия, что позволяет использовать систему без дополнительных затрат. Она поддерживается на всех популярных операционных системах, таких как Linux, macOS и Windows, что делает ее универсальной и доступной для использования в разных условиях. Также PostgreSQL предоставляет все

необходимые инструменты для работы с базами данных прямо «из коробки», не требуя дополнительных программных решений или надстроек.

PostgreSQL выделяется поддержкой различных типов данных и структур, что особенно важно для приложений, работающих с разнообразной информацией, такой как товары, заказы, клиенты и их питомцы. Помимо стандартных типов данных, PostgreSQL поддерживает работу с JSON, геометрическими данными и многими другими типами, что позволяет гибко управлять данными и хранить информацию о товарах и услугах зоомагазина.

Масштабируемость PostgreSQL является важным аспектом для зоомагазина, так как объем данных может увеличиваться по мере роста бизнеса. База данных PostgreSQL может работать с большими объемами данных, ограничиваясь только размерами доступной памяти сервера или облака, что позволяет эффективно масштабировать систему по мере необходимости.

Надежность PostgreSQL обеспечена ее соответствием стандартам ACID, что критически важно для обработки и хранения данных, таких как информация о клиентах и заказах. Это гарантирует целостность и безопасность данных, а также защищает от потерь и сбоев.

Кроме того, PostgreSQL поддерживает многоверсионный контроль параллельного доступа (MVCC), что позволяет нескольким пользователям одновременно работать с базой данных без блокировки. Это особенно важно для зоомагазина, где несколько сотрудников могут одновременно обрабатывать заказы, обновлять информацию о товарах или консультировать клиентов.

Таким образом, PostgreSQL была выбрана для разработки приложения для зоомагазина благодаря своей надежности, функциональности, поддержке различных типов данных и масштабируемости, что делает ее идеальной СУБД для решения задач управления данными в сфере торговли и обслуживания клиентов.

2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Выбор операционной системы

Linux – это операционная система на базе UNIX с открытым исходным кодом. Основным компонентом операционной системы Linux является ядро Linux. Он разработан для предоставления недорогого или бесплатного обслуживания операционной системы пользователям персональных систем, включая систему X-window, редактор Emacs, графический интерфейс IP/TCP и т. д.

Минимальный набор системных библиотек был разработан как часть проекта GNU, с улучшениями, разработанными сообществом Linux. Средства сетевого администрирования Linux были разработаны на основе версии 4.3 Berkeley Software Distribution UNIX. Недавние производные от BSD (например, UNIX FreeBSD), в свою очередь, заимствовали код из Linux.

Система Linux поддерживается слабо связанной сетью разработчиков, взаимодействующих через Internet. Небольшое число публично доступных ftp-серверов используется как хранилища информации о дефакто стандартах.

Стандартный предварительно откомпилированный набор пакетов, или дистрибутивов, включает базовую систему Linux, утилиты для инсталляции системы и управления системой, а также готовые к инсталляции пакеты инструментов для UNIX. Ранние дистрибутивы включали диалекты SLS и Slackware. Red Hat и Debian – популярные дистрибутивы, соответственно, основанные на коммерческих и некоммерческих исходных кодах. [10]

Ядро Linux – один из крупнейших проектов с открытым исходным кодом в мире, код которого вносят тысячи разработчиков, и для каждого выпуска вносятся миллионы строк кода. Оно распространяется под лицензией GPLv2, которая требует, чтобы любая модификация ядра, выполненная в программном обеспечении, поставляемом клиентам, была доступна им (клиентам), хотя на практике большинство компаний делают исходный код общедоступным.

Существует множество компаний (часто конкурирующих), которые вносят свой код в ядро Linux, а также представители научных кругов и независимые разработчики. Текущая модель разработки основана на выпуске релизов через фиксированные промежутки времени (обычно 3–4 месяца). Новые функции добавляются в ядро в течение одной или двух недель. После окна слияния еженедельно создается кандидат на выпуск (rc1, rc2 и т. д.).

Как и любая операционная система, Linux состоит из программного обеспечения, компьютерных программ, документации и аппаратного обеспечения. Основными компонентами операционной системы Linux являются: приложение, оболочка, ядро, оборудование, утилиты. Расположение компонентов можно увидеть на рисунке 2.1.

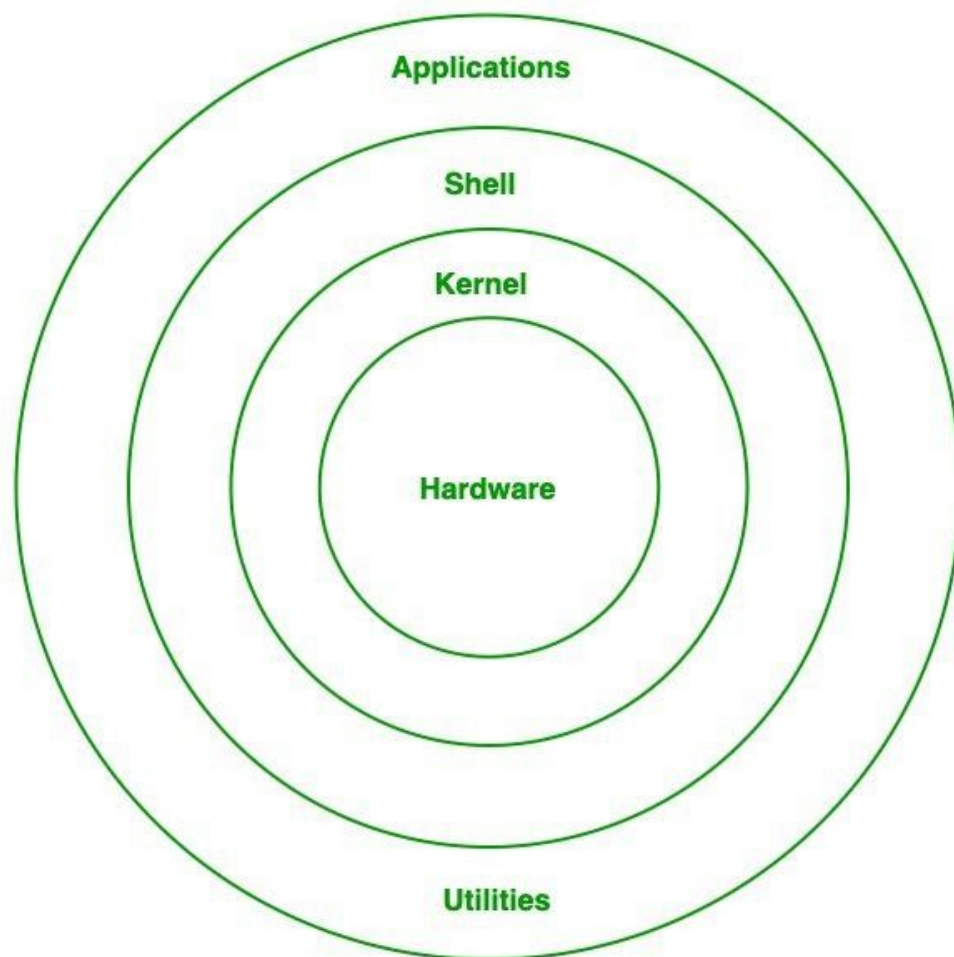


Рисунок 2.1 – Архитектура операционной системы Linux

Ядро (Hardware) – это основной компонент Linux, он контролирует активность других аппаратных компонентов. Он визуализирует общие аппаратные ресурсы и обеспечивает каждый процесс необходимыми виртуальными ресурсами. Это заставляет процесс ждать в очереди готовности и последовательно выполняться, чтобы избежать каких-либо конфликтов.

Существуют различные виды ядер, такие как монолитное, микроядро, экзоядро и гибридное ядро.

Монолитное ядро – это тип ядра операционной системы, в котором все параллельные процессы выполняются одновременно в самом ядре. Все процессы используют одни и те же ресурсы памяти.

В микроядре пользовательские службы и службы ядра выполняются в отдельных адресных пространствах. Пользовательские службы хранятся в адресном пространстве пользователя, а службы ядра – в адресном пространстве ядра.

Exo-kernel предназначен для управления аппаратными ресурсами на уровне приложения. В этой операционной системе используется абстракция высокого уровня, чтобы обеспечить доступ к аппаратным ресурсам ядра.

Это сочетание монолитного ядра и микроядра. Он обладает скоростью и дизайном монолитного ядра, а также модульностью и стабильностью микроядра.

Системные библиотеки (Kernel) – это некоторые предопределенные функции, с помощью которых любые прикладные программы или системные утилиты могут получить доступ к функциям ядра. Эти библиотеки являются основой, на которой может быть построено любое программное обеспечение.

Некоторые из наиболее распространенных системных библиотек: GNU C, libpthread, libdl, libm.

Библиотека GNU C – это библиотека C, которая предоставляет наиболее фундаментальную систему для интерфейса и выполнения программ C. Это обеспечивает множество встроенных функций для выполнения.

Libpthread (POSIX Threads) – библиотека играет важную роль в многопоточности в Linux, она позволяет пользователям создавать и управлять несколькими потоками.

Libdl (динамический компоновщик) – библиотека отвечает за загрузку и связывание файла во время выполнения.

Libm (математическая библиотека) – эта библиотека предоставляет пользователю все виды математических функций и их выполнение.

Некоторые другие системные библиотеки: librt (библиотека реального времени), libcrypt (криптографическая библиотека), libnss (библиотека переключения службы имен), libstdc++ (стандартная библиотека C++).

Оболочка (Shell) – это, в своем роде, интерфейс ядра, который скрывает от пользователя внутреннее выполнение функций ядра. Пользователи могут просто ввести команду и, используя функцию ядра, выполнить конкретную задачу соответствующим образом.

Существует два типа оболочек: оболочка командной строки и графический интерфейс пользователя. Первая выполняет команду,

предоставленную пользователем, указанную в форме команды. Выполняется специальная программа под названием терминал, и результат отображается в самом терминале. Вторая же выполняет процесс, предоставленный пользователем, в графическом виде, а выходные данные отображаются в графическом окне.

На рисунке 2.2 представлено графическое отображение оболочки Linux.

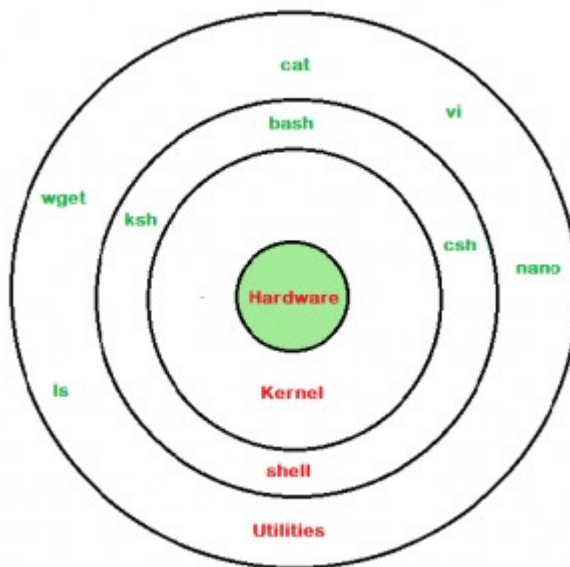


Рисунок 2.2 – Оболочка Linux

Аппаратный уровень Linux (Applications) – это самый нижний уровень операционной системы. Он играет жизненно важную роль в управлении всеми аппаратными компонентами. Он включает в себя драйверы устройств, функции ядра, управление памятью, управление процессором и операции ввода-вывода. Этот уровень обобщает высокую сложность, предоставляя интерфейс для программного обеспечения и гарантируя правильную функциональность всех компонентов.

Системные утилиты (Utilities) – это инструменты командной строки, которые выполняют различные задачи, предоставляемые пользователем, чтобы улучшить управление и администрирование системы. Эти утилиты позволяют пользователю выполнять различные задачи, такие как управление файлами, мониторинг системы, настройка сети, управление пользователями и т. д.

Самым мощным и универсальным компонентом операционной системы Linux является ядро, с помощью которого можно управлять функционалом всей операционной системы. Ядро предоставляет огромный

набор функций, к которым пользователь может легко получить доступ в интерактивном режиме с помощью оболочки. Для правильной работы операционной системы необходимо подходящее оборудование. Все компоненты операционной системы делают ее проще, быстрее, стабильнее и надежнее. [11]

У операционной системы Linux существует множество плюсов для решения типичных задач программирования. Однако стоит выделить основные.

Если сравнивать с проприетарными ОС, то главный плюс GNU/Linux, как и десятка других свободных ОС (например, Free/Net/OpenBSD, OpenIndiana), это то, что они являются свободным ПО. Это означает что, пользователь может без ограничений запускать и использовать эти ОС для любых целей, изучать и модифицировать их работу. А также помогать окружающим, распространяя копии ОС и их модификации.

Широкая поддержка аппаратного обеспечения. Много драйверов для устройств, особенно на домашних системах (где дешевые не серверные компоненты). Есть вероятность что какое-либо оборудование не будет поддерживаться в системе типа BSD или OpenIndiana.

Многие дистрибутивы GNU/Linux могут работать на старых компьютерах гораздо лучше, чем системы типа Windows или macOS. Они зачастую могут вообще отказаться на них работать.

Активная поддержка пользователей. За десятилетия существования у GNU/Linux образовался круг пользователей и разработчиков, которые смогут оперативно помочь с задачами или проблемами, возникающими при работе у неопытных пользователей.

Плюсом персонально для разработчиков является возможность переделать ОС под ваши задачи. Можно доработать как всю систему, так и отдельные ее компоненты, найти и исправлять недочеты или нанять разработчиков для этих задач. С несвободным программным обеспечением, есть только надежда, что компания владелец ПО соизволит выполнить ваше желание, еще и за вменяемый срок. А также возможность делиться своими наработками и исправлениями. [12]

2.2 Выбор платформы для написания программы

Python является одним из наиболее популярных и универсальных языков программирования, и его идеальность для программирования проявляется во многих аспектах, таких как синтаксис. Python прост и

лаконичен, что делает его легким для изучения и использования. Это способствует написанию чистого и читаемого кода, что упрощает поддержку и совместную работу. Python имеет огромное количество стандартных библиотек, покрывающих практически все аспекты разработки, начиная от обработки данных и веб-разработки, и заканчивая машинным обучением и искусственным интеллектом. Это позволяет разработчикам быстро создавать функциональные приложения без необходимости писать код с нуля. Также он является языком с динамической типизацией, что означает, что разработчики могут создавать и изменять переменные без необходимости объявления типа заранее. Это упрощает процесс разработки и делает код более гибким. И нельзя забыть огромное и активное сообщество разработчиков, которые создают и поддерживают библиотеки, фреймворки и инструменты для разработки. Это обеспечивает доступ к большому объему ресурсов, документации и поддержки, что помогает разработчикам быстро решать проблемы и улучшать свои навыки.

Для написания кода будет использоваться среда разработки PyCharm.

PyCharm – это программное обеспечение, которое представляет собой интегрированную среду разработки для языка программирования Python.

Использование IDE PyCharm позволяет значительно ускорить разработку программного обеспечения, минимизировать ошибки и облегчить сотрудничество между программистами. Благодаря широкому спектру возможностей, эта интегрированная среда разработки помогает создавать качественное ПО в короткие сроки.

К основным возможностям PyCharm относятся:

1 Интеллектуальное автодополнение кода. PyCharm анализирует код и предлагает варианты автодополнения, которые соответствуют контексту. Это значительно ускоряет написание кода и уменьшает количество ошибок.

2 Проверка кода. PyCharm проверяет код на наличие ошибок и потенциальных проблем, выделяя их прямо в редакторе. Это помогает писать более качественный и надежный код.

3 Отладка. PyCharm включает в себя мощный отладчик, который позволяет пошагово выполнять код, устанавливать точки останова и исследовать значения переменных.

4 Рефакторинг. PyCharm предлагает различные инструменты рефакторинга, которые помогут легко изменить структуру вашего кода, не нарушая его функциональность.

5 Интеграция с системами контроля версий. PyCharm интегрируется с популярными системами контроля версий, такими как Git, Mercurial и SVN, что упрощает управление вашим кодом.

6 Поддержка различных фреймворков. PyCharm предоставляет поддержку популярных Python-фреймворков, таких как Django, Flask и Pyramid.

7 Множество плагинов. PyCharm имеет богатую экосистему плагинов, которые расширяют его функциональность.

3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

3.1 Обоснование необходимости разработки

Современные зоомагазины сталкиваются с рядом вызовов, связанных с эффективным управлением товарами, заказами и обслуживанием клиентов. В условиях роста конкуренции и изменяющихся потребностей потребителей, необходима система, которая обеспечит удобство и оперативность в обслуживании, а также повысит общую эффективность работы магазина.

С каждым годом все больше людей предпочитают совершать покупки через интернет, включая покупку товаров для своих питомцев. Веб-приложение для зоомагазина позволяет клиентам быстро и удобно выбирать товары, делать заказы и оплачивать их без необходимости посещать физический магазин. Это особенно важно для людей, у которых нет времени на походы в магазин, а также для тех, кто ищет редкие или специализированные товары, которые могут быть недоступны в обычных зоомагазинах.

В зоомагазинах часто обновляется ассортимент товаров, и управление ими вручную становится сложной задачей. Веб-приложение позволит автоматизировать процессы учета товарных запасов, обновления данных о наличии товаров и их цен. Это улучшит процессы инвентаризации и предотвратит ошибки, связанные с некорректной информацией о товаре.

Веб-приложение будет доступно с любого устройства, что позволяет владельцам и сотрудникам зоомагазина иметь доступ к информации в любое время и из любого места. Это повышает гибкость работы магазина и позволяет эффективно управлять бизнесом, независимо от физического местоположения. В целях повышения удобства и доступности для пользователей будет разработан интуитивно понятный интерфейс, который позволит быстро находить нужные товары, оформлять заказы и следить за статусом доставки. Также будет предусмотрена система фильтров и поиска для удобства работы с большим ассортиментом товаров. Разработка данного веб-приложения необходима для обеспечения конкурентоспособности зоомагазина в условиях быстро развивающегося рынка и изменения потребительских предпочтений. Это позволит не только улучшить взаимодействие с клиентами, но и повысить эффективность работы бизнеса в целом.

3.2 Технологии программирования, используемые для решения поставленных задач

В качестве языка программирования для написания программы используется Python.

Python – высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью, ориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение переносимости написанных на нём программ. Язык является полностью объектно-ориентированным в том плане, что всё является объектами.

Синтаксис ядра языка минималистичен, за счёт чего на практике редко возникает необходимость обращаться к документации. Сам же язык известен как интерпретируемый и используется в том числе для написания скриптов[. Недостатками языка являются зачастую более низкая скорость работы и более высокое потребление памяти написанных на нём программ по сравнению с аналогичным кодом, написанным на компилируемых языках, таких как C или C++.

Python предоставляет удобные инструменты для отладки и разработки, что облегчает создание и отладку кода, работающего с потоками и процессами.

Остановимся также на реализации графического интерфейса. Графический интерфейс приложения для дистанционного пульта управления будет реализован с помощью использования фреймворка Django.

Django – это платформа Python, которая упрощает создание веб-сайтов с использованием Python. Django берет на себя все сложные задачи, чтобы мы могли сосредоточиться на веб-приложении. Django подчеркивает возможность повторного использования компонентов, также называемую DRY (не повторяйте себя), и поставляется с готовыми к использованию функциями, такими как система входа в систему, подключение к базе данных и операции CRUD (создать, прочитать, обновить, удалить).

С Django мы можем реализовать веб-приложения от концепции до запуска за считанные часы. Django берет на себя большую часть хлопот веб-разработки, поэтому мы можем сосредоточиться на написании своего приложения, не изобретая велосипед. Это бесплатно и с открытым исходным кодом. Также Django был разработан, чтобы помочь разработчикам максимально быстро доводить приложения от концепции до завершения. Он включает в себя десятки дополнительных возможностей, которые можно

использовать для решения распространенных задач веб-разработки. Django заботится об аутентификации пользователей, администрировании контента, картах сайта, RSS-каналах и многих других задачах — прямо из коробки. К тому же Django серьезно относится к безопасности и помогает разработчикам избежать многих распространенных ошибок безопасности, таких как внедрение SQL, межсайтовый скриптинг, подделка межсайтовых запросов и кликджекинг. Его система аутентификации пользователей обеспечивает безопасный способ управления учетными записями пользователей и паролями. Некоторые из самых загруженных сайтов на планете используют способность Django быстро и гибко масштабироваться для удовлетворения самых высоких требований к трафику. Компании, организации и правительства использовали Django для создания самых разных вещей — от систем управления контентом до социальных сетей и платформ научных вычислений.

Django следует шаблону проектирования MVT (шаблон представления модели):

1 Модель — данные, которые вы хотите представить, обычно данные из базы данных.

В Django данные доставляются в виде объектно-реляционного сопоставления (ORM) — метода, предназначенного для упрощения работы с базами данных. Наиболее распространенным способом извлечения данных из базы данных является SQL. Одна из проблем SQL заключается в том, что вам нужно довольно хорошо понимать структуру базы данных, чтобы иметь возможность работать с ней. Django с ORM упрощает взаимодействие с базой данных без необходимости писать сложные операторы SQL. Модели обычно находятся в файле `models.py`.

2 Представление — обработчик запроса, который возвращает соответствующий шаблон и контент — на основе запроса пользователя.

Представление — это функция или метод, который принимает HTTP-запросы в качестве аргументов, импортирует соответствующие модели, определяет, какие данные отправлять в шаблон, и возвращает окончательный результат. Представления обычно находятся в файле с именем `views.py`.

3 Шаблон — текстовый файл (например, HTML-файл), содержащий макет веб-страницы с логикой отображения данных.

Шаблоны часто представляют собой файлы `.html` с HTML-кодом, описывающим макет веб-страницы, но они также могут быть в других форматах файлов для представления других результатов, но мы сосредоточимся на файлах `.html`. Django использует стандартный HTML для

описания макета, но для добавления логики использует теги Django. Модель предоставляет данные из базы данных. Шаблоны приложения находятся в папке templates.

Плюсами Django являются:

1 Django предоставляет множество встроенных инструментов и библиотек, которые значительно ускоряют процесс разработки веб-приложений. Это включает в себя автоматическую генерацию административного интерфейса, систему аутентификации пользователей, формы для ввода данных и многое другое.

2 Django поставляется с встроенным ORM (Object-Relational Mapping), который позволяет работать с базой данных, используя объектно-ориентированный подход. Это делает взаимодействие с базой данных более интуитивным и удобным, а также обеспечивает безопасность от SQL-инъекций.

3 Django создан для работы с проектами любого размера – от небольших веб-сайтов до крупных веб-приложений. Его модульная структура и гибкий архитектурный подход делают возможным легкое масштабирование приложений по мере их роста.

4 Django имеет активное сообщество разработчиков, которое предоставляет обширную документацию, учебные ресурсы, плагины и расширения. Это обеспечивает поддержку и помощь в решении проблем.

5 Django предоставляет множество встроенных механизмов для обеспечения безопасности веб-приложений. Это включает в себя защиту от атак на основе параметров URL (Cross Site Scripting, XSS), защиту от инъекций SQL, защиту от подделки запросов межсайтовых запросов (Cross Site Request Forgery, CSRF) и многое другое.

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

4.1 Подключение к базе данных

Для взаимодействия с базой данных PostgreSQL в проекте используется асинхронный пул подключений, реализованный с помощью библиотеки `asyncpg`. Это позволяет эффективно управлять подключениями и выполнять запросы к базе данных в условиях высокой нагрузки.

Подключение к базе данных организовано через класс `Database`, который содержит методы для установки и завершения соединения.

Метод `connect()` отвечает за инициализацию пула подключений. В процессе выполнения устанавливается соединение с базой данных с параметрами: пользователь, пароль, имя базы данных, адрес хоста, порт.

Для повышения производительности и стабильности подключения используются настройки минимального (`min_size=1`) и максимального (`max_size=10`) количества подключений в пуле. При успешной инициализации выводится сообщение об успешном подключении к базе данных. В случае возникновения ошибки выводится информация об ошибке.

Метод `disconnect()` закрывает пул подключений, обеспечивая корректное завершение взаимодействия с базой данных.

Для взаимодействия с базой данных PostgreSQL в классе `Database` реализованы универсальные методы, позволяющие выполнять запросы различного типа. Метод `execute` предназначен для выполнения запросов, которые не возвращают данных, таких как `INSERT`, `UPDATE` или `DELETE`. Он принимает SQL-запрос в виде строки и дополнительные аргументы для подстановки значений, а выполнение осуществляется с использованием соединения из пула, что гарантирует эффективное использование ресурсов. Метод `fetch` используется для получения списка строк в результате выполнения запроса, например, `SELECT`. Каждая строка преобразуется в словарь, где ключами являются имена столбцов, а результатом выполнения является список таких словарей, что удобно для дальнейшей обработки данных.

Метод `fetchrow` предназначен для получения одной строки из результата запроса. Он возвращает данные в виде словаря, если строка существует, или `None`, если запрос не вернул данных. Это подходит для

случаев, когда необходимо извлечь информацию о конкретной записи, например, данные пользователя или товара. Метод `fetchval` возвращает одно значение из первой строки результата запроса. Этот подход удобен, если необходимо получить, например, количество записей в таблице или значение определенного столбца. Если данных в запросе нет, метод возвращает `None`.

Все методы используют конструкцию `async with` для управления соединениями из пула. Это позволяет гарантировать автоматическое возвращение соединения в пул после выполнения запроса, даже в случае возникновения ошибки, что делает приложение более устойчивым и безопасным. Такой подход обеспечивает удобство работы с базой данных и упрощает реализацию взаимодействия с PostgreSQL в асинхронной среде.

4.2 Регистрация и авторизация пользователей

Для обеспечения безопасного доступа к функционалу интернет-магазина разработан модуль регистрации и авторизации пользователей, реализованный с использованием фреймворка FastAPI. Основной задачей данного модуля является создание, хранение и управление учетными записями пользователей, а также предоставление доступа к функционалу в зависимости от их роли. Для реализации авторизации используется JWT (JSON Web Token).

Процесс регистрации включает передачу пользователем логина, пароля, имени (опционально) и автоматическое назначение базовой роли. Перед сохранением данных в таблице `Users` базы данных PostgreSQL производится проверка уникальности логина и пароля.

Авторизация осуществляется через предоставление пользователем логина и пароля. После успешной проверки пользовательских данных генерируется JWT токен, содержащий ID пользователя, его роль и время истечения действия токена. Этот токен возвращается клиенту и используется для доступа к защищенным ресурсам.

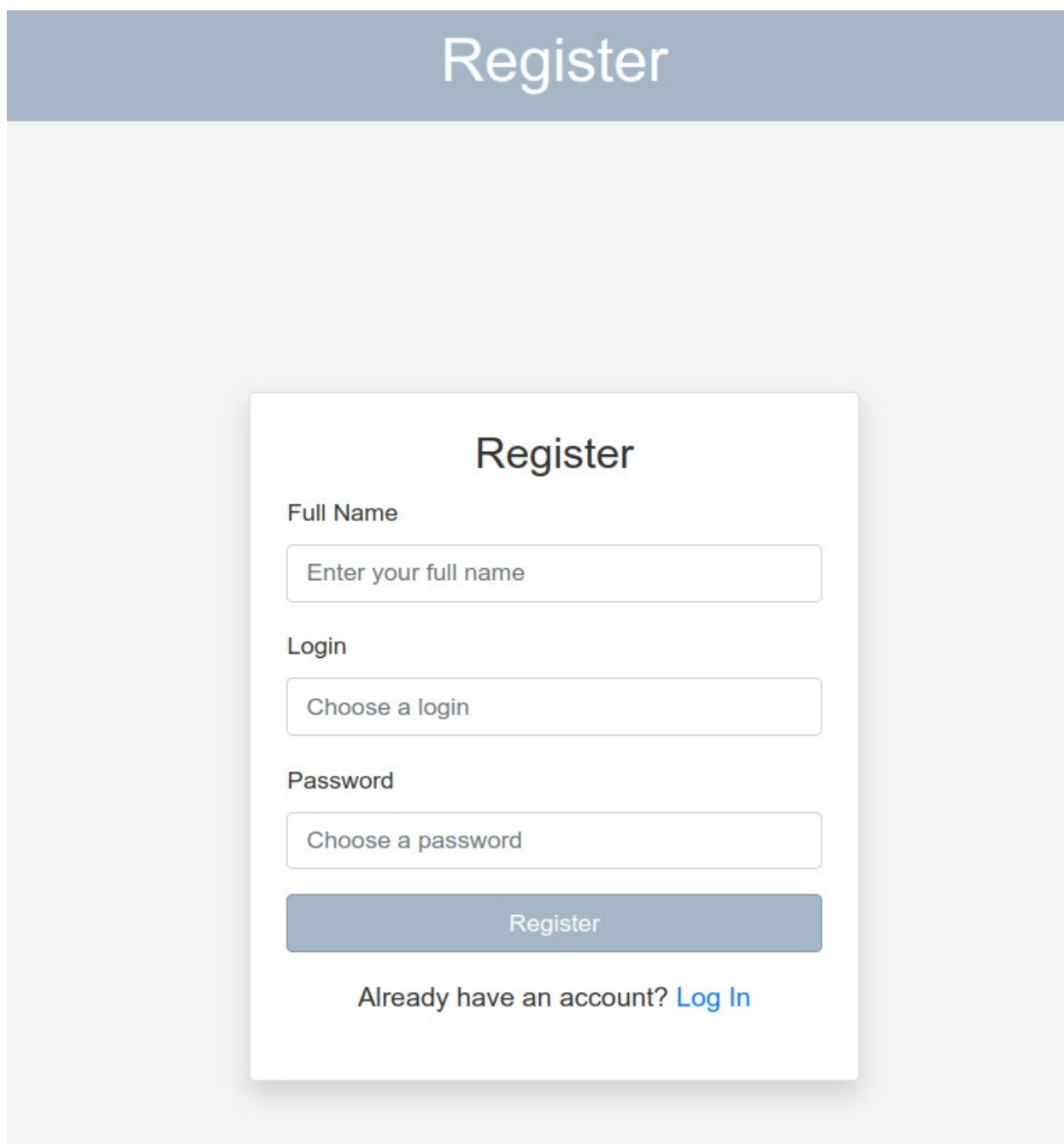
JWT токен передается в заголовке `Authorization` в формате `Bearer <token>`. На стороне сервера реализовано middleware, проверяющее наличие токена, его валидность и права доступа пользователя.

Таблица `Users` в PostgreSQL содержит следующие поля: `Id` (уникальный идентификатор), `Login` (уникальный логин), `Password`

(хэшированный пароль), Name (имя), RoleId (роль), CouponId (идентификатор купона), Banned (флаг блокировки).

Модуль обеспечивает безопасную регистрацию и аутентификацию пользователей, а также позволяет разделять доступ к ресурсам в зависимости от ролей.

На рисунке 4.1 представлен интерфейс для регистрации пользователя.



The image shows a web registration form. At the top, there is a blue header bar with the word "Register" in white. Below this, the main content area is light gray. In the center, there is a white rectangular box with a subtle shadow, also titled "Register". Inside this box, there are three input fields: "Full Name" with the placeholder "Enter your full name", "Login" with the placeholder "Choose a login", and "Password" with the placeholder "Choose a password". Below these fields is a blue button labeled "Register". At the bottom of the white box, there is a link that says "Already have an account? Log In".

Рисунок 4.1 – Регистрация пользователя

Далее на рисунке 4.2 представлен пользовательский интерфейс для авторизации пользователей системы.

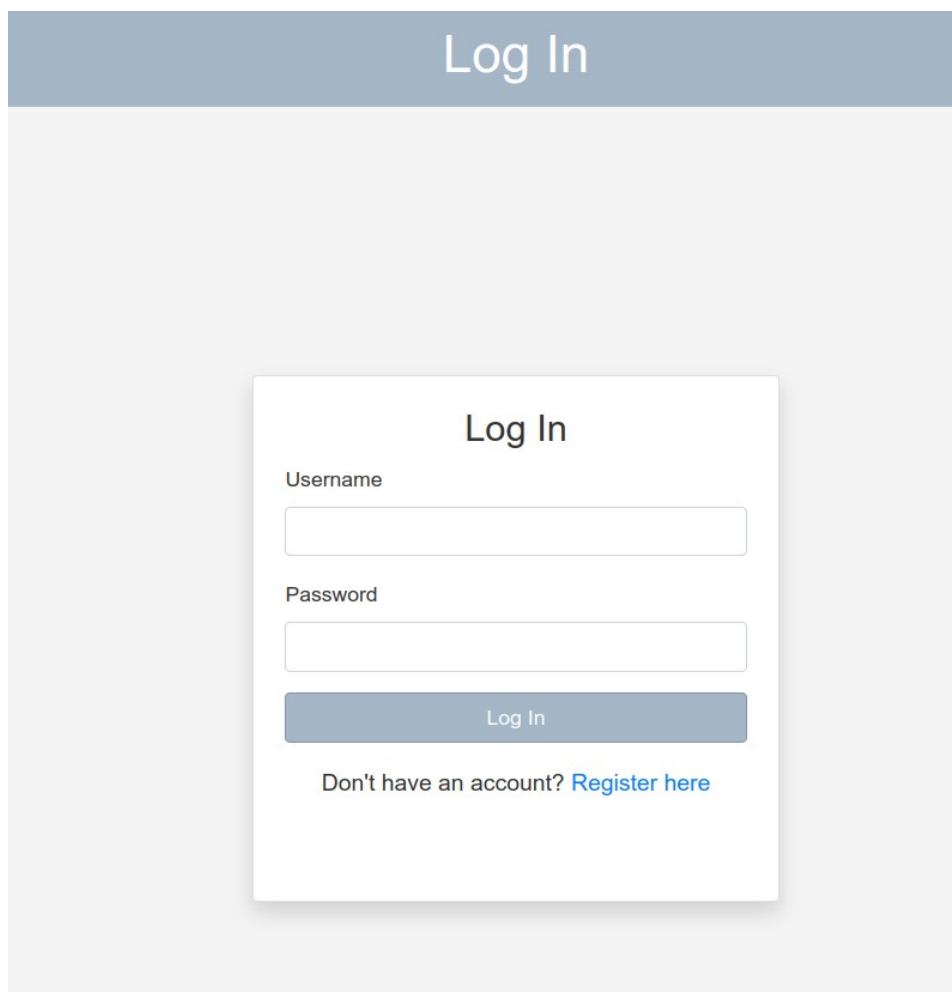
The image shows a user login interface. At the top, there is a dark blue header bar with the text "Log In" in white. Below this, the main background is a light gray. In the center, there is a white rectangular box with a subtle shadow. Inside this box, the title "Log In" is centered at the top. Below the title, there are two input fields: the first is labeled "Username" and the second is labeled "Password". Both labels are in a small, dark gray font. Below the password field, there is a dark blue button with the text "Log In" in white. At the bottom of the white box, there is a link that says "Don't have an account? [Register here](#)", where "Register here" is in blue text.

Рисунок 4.2 – Авторизация пользователя

4.3 Управление пользователями

Управление пользователями в системе организовано с учетом разграничения прав доступа между обычными пользователями и администраторами. Обычный пользователь имеет возможность изменять информацию в своем профиле, такую как имя и другие личные данные, а также выполнять вход в систему и выход из аккаунта. Авторизация пользователя осуществляется с помощью токенов JWT, которые обеспечивают безопасный доступ к защищенным ресурсам. При выходе пользователя из аккаунта токен становится недействительным, что предотвращает его дальнейшее использование.

Администратор обладает расширенными правами и может управлять учетными записями всех пользователей системы. Это включает просмотр списка всех зарегистрированных пользователей с их основными данными, а также возможность изменения их состояния. Администратор может заблокировать учетную запись пользователя, тем самым ограничивая его доступ к системе, либо снять блокировку, восстанавливая его права.

Дополнительно в системе реализован механизм логирования всех действий пользователей. Каждое изменение в системе, будь то обновление профиля, смена пароля, вход или выход из аккаунта, фиксируется в журнале действий. Администратор имеет доступ к этому журналу и может просматривать как общую историю изменений, так и изменения, связанные с конкретным пользователем. Это позволяет обеспечивать прозрачность и контроль за действиями в системе, а также оперативно реагировать на подозрительные действия или нарушения.

Таким образом, управление пользователями реализовано с учетом удобства для конечных пользователей и полного контроля для администраторов, что обеспечивает безопасность и устойчивость системы.

4.4 Взаимодействие с сущностями приложения

В приложении для зоомагазина реализована функциональность взаимодействия с сущностями приложения, которая учитывает потребности как обычных пользователей, так и администраторов. Для пользователей предусмотрена возможность просматривать весь каталог товаров, представленных в магазине. Это позволяет ознакомиться с ассортиментом и выбрать интересующие позиции. Каждый товар имеет персонализированную карточку, содержащую расширенную информацию, такую как описание, характеристики, цена и наличие на складе.

Для управления покупками пользователю предоставлена корзина, которая автоматически создается в момент регистрации и закрепляется за учетной записью. Пользователь может добавлять товары в корзину, где они хранятся до момента оформления заказа или удаления. В корзине доступна функция удаления ненужных товаров, а также перемещение товаров в заказ, который фиксирует намерение пользователя получить выбранные позиции в физическом магазине. Это упрощает процесс планирования покупок и взаимодействие с магазином.

Кроме того, пользователи могут искать и просматривать товары, отсортированные по ключевым параметрам, таким как фирма-производитель,

категория или вид животного, для которого предназначен товар. Это позволяет быстро находить необходимые позиции и облегчает выбор.

Администратор обладает расширенными правами, что позволяет ему выполнять важные задачи по управлению содержимым магазина и обеспечению его бесперебойной работы. В его распоряжении находится функционал создания, изменения и удаления товаров, благодаря которому ассортимент магазина может оперативно обновляться. Администратор также может управлять справочными данными, такими как список фирм-производителей, категории товаров и виды животных. Возможность добавления, редактирования или удаления этих данных обеспечивает актуальность и структурированность информации в системе.

Особое внимание уделено управлению заказами. Администратор имеет доступ к полному списку заказов, оформленных пользователями, что позволяет ему контролировать процесс обработки заявок, организовывать выдачу товаров в физическом магазине и отслеживать выполненные и невыполненные заказы. Это создает основу для эффективного взаимодействия между онлайн-платформой и офлайн-магазином.

На рисунке 4.3 представлено отображение всех товаров системы.

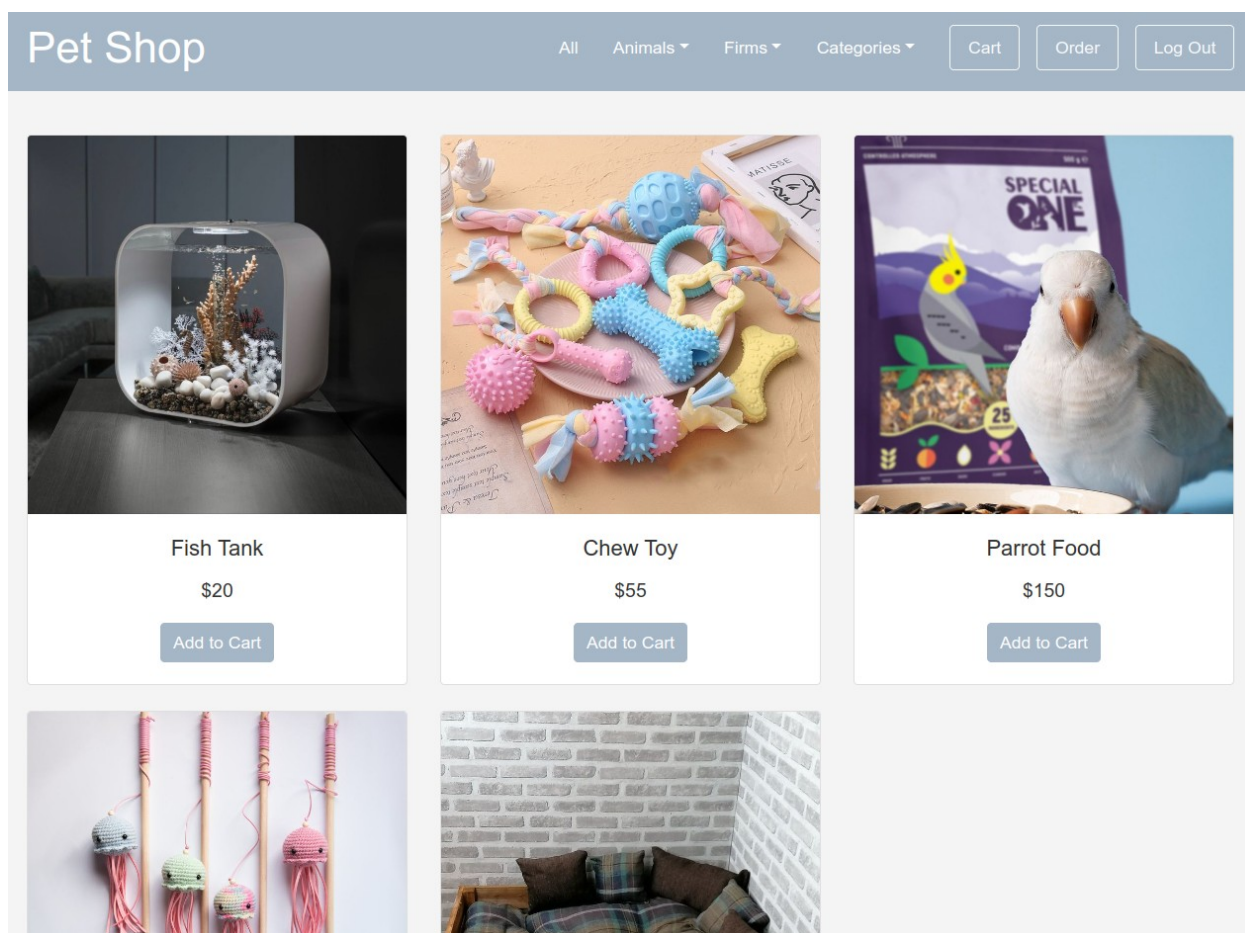


Рисунок 4.3 – Отображение товаров

На рисунке 4.4 и 4.5 представлен интерфейс для администрации, что позволяет изменять основные сущности приложения.

На 4.4 представлено отображение всех возможных фирм и категорий. Каждую фирму и категорию можно удалить, также дана возможность ввода в поле создания новой фирмы или категории.

Admin panel

Home

Firms

Acana	<input type="button" value="Delete"/>
Zoo	<input type="button" value="Delete"/>
Fir-fir	<input type="button" value="Delete"/>
Xoo	<input type="button" value="Delete"/>

Categories

Toys	<input type="button" value="Delete"/>
Furniture	<input type="button" value="Delete"/>
Accessories	<input type="button" value="Delete"/>

Рисунок 4.4 – Отображение изменения сущностей

На рисунке 4.5 представлен оставшийся интерфейс, а именно добавление и удаление животного и добавление, удаление и изменение заказа. Также есть возможность изменить заказ при нажатии на Edit: открывается дополнительное меню с полями ввода обновляемых данных.

Animals

Animal Type	Add Animal
Cat	Delete
Parrot	Delete
Fish	Delete

Add good

Good title:

ID Firm:

ID Category:

ID Animal:

Price:

URL picture:

Add good

Рисунок 4.5 – Отображение изменения сущностей

Также основной сущностью системы являются корзина и заказ, которые играют ключевую роль в процессе взаимодействия пользователя с магазином. Корзина создается автоматически при регистрации пользователя и является его персональным инструментом для управления товарами перед покупкой. В корзину можно добавлять любое количество товаров, просматривать их список, удалять ненужные позиции или корректировать выбор.

Когда пользователь завершает формирование корзины, он может переместить выбранные товары в заказ. Заказ фиксирует намерение пользователя приобрести указанные позиции и получить их в офлайн-магазине. После оформления заказа его данные становятся доступными администратору, который отслеживает статус выполнения заявок. Это позволяет обеспечить оперативную обработку заказов, их сборку и выдачу клиенту.

Корзина и заказ интегрированы в общую архитектуру приложения, что делает процесс покупок интуитивно понятным и удобным для пользователя.

Одновременно с этим администратор получает полный контроль над оформленными заказами, что позволяет эффективно организовать работу офлайн-магазина, своевременно подготавливать товары для выдачи и поддерживать высокий уровень сервиса. Такой подход обеспечивает целостность системы и удобство взаимодействия для всех категорий пользователей.

На рисунке 4.6 представлена корзина пользователя с добавленными в нее товарами.

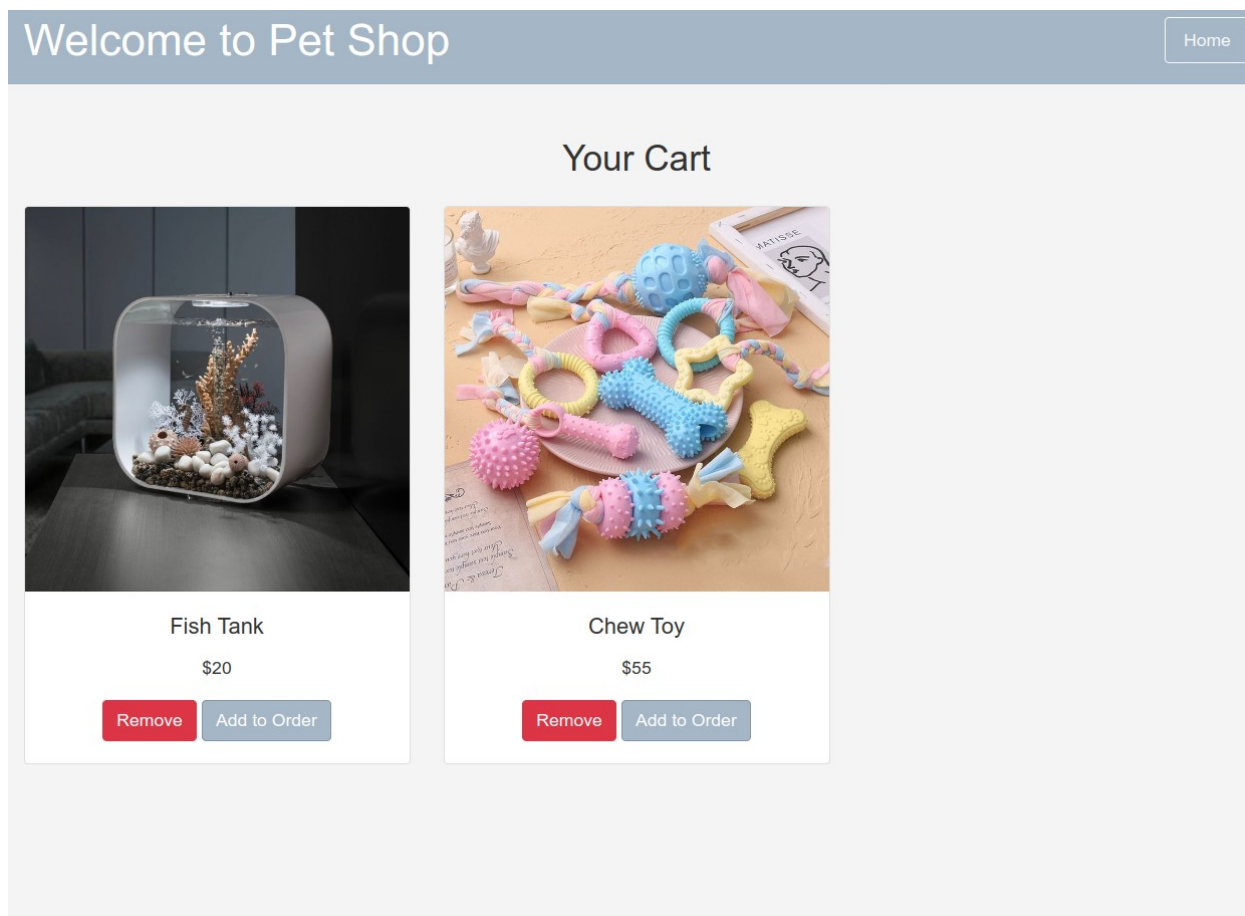


Рисунок 4.6 – Отображение корзины

Далее на рисунке 4.7 представлен интерфейс заказа пользователя.

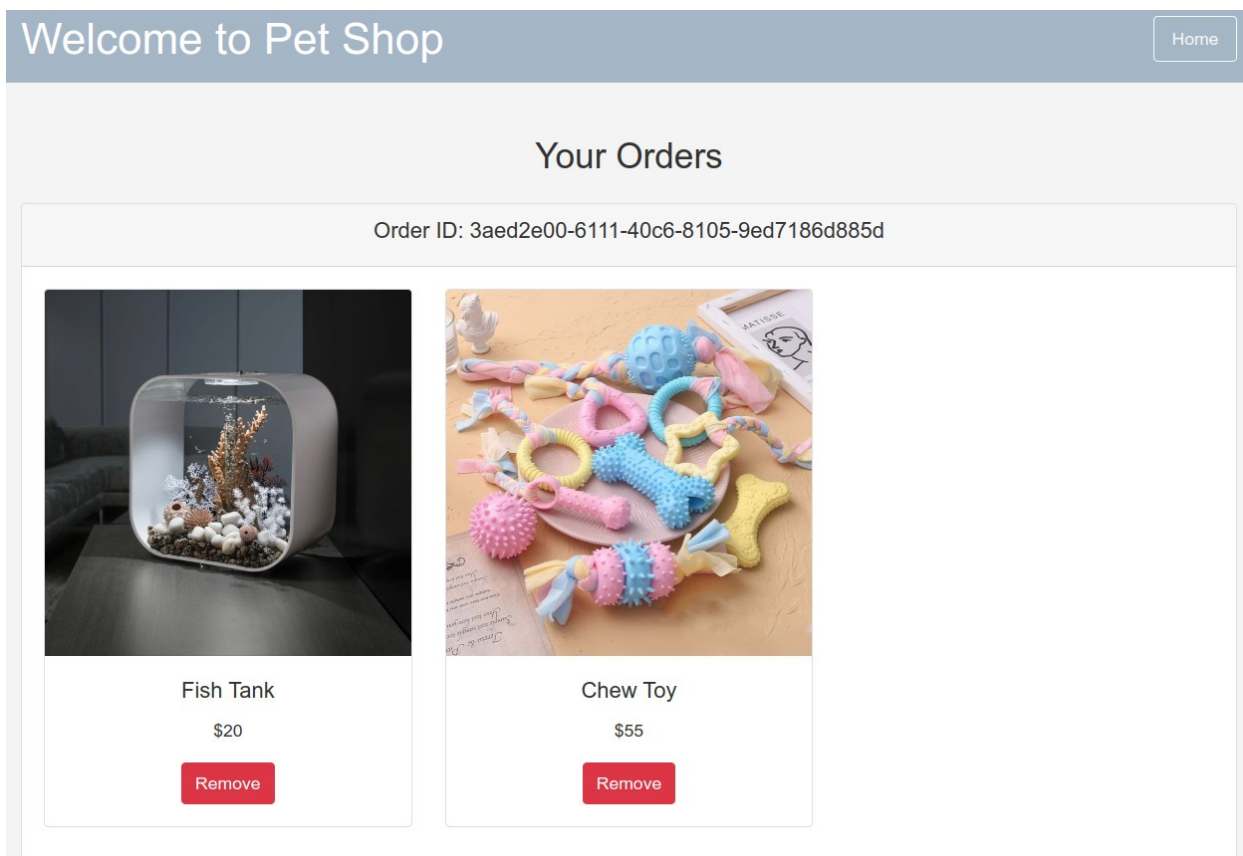


Рисунок 4.7 – Отображение заказа

Также на рисунке 4.8 отображается интерфейс, доступный только администратору, что видит все заказы системы.

Orders			
Home			
Order ID	User ID	Goods	Action
3aed2e00-6111-40c6-8105-9ed7186d885d	26a26b25-92f8-415c-b21d-d55b59d6a215	17799039-1d9c-40e0-9328-0439a1927a0b, 9293c290-7497-4db0-aba1-6a4e95e83c78	Remove

Рисунок 4.8 – Отображение всех заказов

4.6 Руководство пользователя

Система представляет собой интернет-магазин, разработанный для зоомагазина, который предоставляет пользователям удобные инструменты для поиска, выбора и заказа товаров, а администраторам — мощные средства управления ассортиментом и пользователями. Основная цель системы —

упростить взаимодействие клиентов с магазином и обеспечить эффективное управление процессами.

После регистрации в системе пользователь получает доступ к функционалу просмотра каталога товаров. На главной странице представлены карточки товаров с краткой информацией, такие как название, цена и изображение. Для более детального ознакомления можно открыть карточку товара, где представлены описание, характеристики, фирма-производитель, категория и рекомендации по использованию.

Корзина, которая автоматически создается при регистрации, позволяет пользователю собирать и управлять выбранными товарами. Товары можно добавлять, удалять или перемещать в заказ для оформления получения в офлайн-магазине. Заказы формируются из содержимого корзины и фиксируют список товаров, которые пользователь планирует приобрести. Кроме того, товары можно фильтровать и сортировать по фирме, виду животного и категории, что упрощает поиск нужной продукции.

Администратор имеет доступ к расширенному функционалу. Он может создавать, редактировать и удалять товары, управлять списками фирм, животных и категорий, а также отслеживать все заказы в системе. Администратор имеет возможность управлять учетными записями пользователей, включая блокировку и разблокировку, и просматривать журнал действий. Все действия пользователей, включая изменения профиля, операции с корзиной, создание заказов и административные операции, фиксируются в таблице логов, что обеспечивает прозрачность и контроль над процессами.

База данных системы реализована на PostgreSQL и включает следующие основные таблицы:

Таблица **Users** содержит учетные данные пользователей, включая логин, пароль (в зашифрованном виде), имя, роль, состояние (например, блокировку) и привязанные данные. Таблица **Roles** служит для управления ролями пользователей, такими как обычный пользователь и администратор, и содержит информацию о правах доступа. Таблица **Goods** хранит информацию о товарах: название, описание, цену, категорию, фирму-производителя и доступность. Таблица **Cart** связывает пользователей с товарами, добавленными в корзину. Таблица **Orders** содержит данные о заказах, включая их статус, список товаров и пользователя, оформившего заказ.

Таблица `Logs` фиксирует все изменения и действия, выполненные пользователями или администраторами, с привязкой ко времени, пользователю и описанию действия, что позволяет администратору отслеживать все операции и изменения в системе. Таблицы `Categories`, `Animals` и `Firms` представляют собой справочные данные, структурирующие информацию о категориях товаров, видах животных и фирмах соответственно.

Запросы к базе данных выполняются через эндпоинты, реализованные на FastAPI. Эндпоинты используют методы взаимодействия с базой данных, такие как `execute`, `fetch`, `fetchrow` и `fetchval`. Каждый эндпоинт выполняет определенную функцию, например, получение списка товаров, добавление товара в корзину, создание заказа или изменение профиля.

Результаты запросов возвращаются в формате, удобном для обработки на клиентской стороне. Это могут быть структурированные данные в формате `JSON`, представляющие содержимое таблиц, либо предопределенные схемы данных, которые формируются с использованием библиотек сериализации, таких как `Pydantic`. Такой подход обеспечивает консистентность и удобство работы с данными, а также упрощает интеграцию фронтенда и бекенда.

Система проста в использовании и обладает интуитивно понятным интерфейсом. Пользователь может быстро зарегистрироваться, авторизоваться и приступить к выбору товаров. Администратор, в свою очередь, имеет доступ ко всем ключевым функциям управления, что делает систему удобной и функциональной для всех категорий пользователей.

5 ПРОЕКТИРОВАНИЕ РАЗРАБАТЫВАЕМОЙ БАЗЫ ДАННЫХ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Данная база данных предназначена для системы управления зоомагазином. Она организована таким образом, чтобы поддерживать ключевые бизнес-процессы: управление товарами, категориями, заказами, корзинами, пользователями, ролями и купонами. В базе данных определено 10 сущностей, которые связаны между собой через различные типы отношений.

5.1 Разработка информационной модели

Процесс разработки информационной модели базы данных для программного обеспечения зоомагазина включал несколько этапов, направленных на создание оптимальной структуры данных, обеспечивающей эффективное хранение и обработку информации.

При проектировании базы данных для зоомагазина был проведен анализ существующих онлайн-магазинов, чтобы понять основные требования к системе и эффективные модели данных. Подобные сайты используют базы данных, которые включают сущности, такие как товары, пользователи, заказы, корзины, а также систему скидок и акций. Все эти элементы связаны через реляционные таблицы, где товары классифицируются по категориям, пользователи могут создавать заказы и управлять корзинами, а купоны предоставляют скидки на покупки.

На следующем этапе было определено, какие данные необходимо хранить в системе. Были выделены ключевые сущности, такие как товары, фирмы, категории товаров, типы животных, корзина, заказы, пользователи и их действия, роли и купоны. Каждая сущность была тщательно изучена, чтобы определить ее характеристики и взаимодействия с другими объектами.

На этапе разработки модели было уделено особое внимание обеспечению корректной структуры данных, минимизации избыточности и исключению аномалий. Каждая сущность была нормализована с целью обеспечения целостности данных и предотвращения дублирования информации. Например, для предотвращения аномалий в данных были использованы ключи и ограничения целостности, такие как уникальные индексы, внешние ключи и обязательные поля. Это обеспечивало правильное связывание данных и минимизацию ошибок при их изменении или удалении.

Конкретная структура таблиц была выбрана с учетом специфики системы и потребностей зоомагазина. Например, связь "многие ко многим" между заказами и товарами, реализуемая через вспомогательную таблицу "OrderGood", была выбрана для того, чтобы точно отслеживать, какие товары были заказаны в рамках конкретного заказа. Такая структура позволяет эффективно работать с большими объемами данных, а также легко расширять функциональность системы. Использование вспомогательных таблиц для реализации сложных связей между сущностями, таких как "CartGood", позволяет минимизировать дублирование информации и улучшает гибкость системы при добавлении новых функций.

В результате, данная структура таблиц соответствует задачам системы и позволяет эффективно управлять данными о товарах, заказах, пользователях и купонах, обеспечивая при этом минимизацию избыточности и исключение аномалий. Спроектированная информационная модель позволяет системе работать быстро и слаженно, поддерживая высокую степень масштабируемости и надежности.

5.2 ER-диаграмма базы данных

ER-диаграмма базы данных (диаграмма «сущность-связь») представляет собой визуальное отображение структуры базы данных, где схематично показаны основные сущности, их ключевые атрибуты, а также связи, которые образуют логику взаимодействия между ними. Такие диаграммы служат основой для проектирования и понимания того, как будет организована и функционировать база данных в рамках системы. Важной составляющей ER-диаграммы является наличие различных типов связей между сущностями, таких как «один к одному», «один ко многим» и «многие ко многим», что позволяет адекватно отразить взаимоотношения в реальной предметной области.

С помощью ER-диаграммы можно наглядно понять, как данные будут храниться и обрабатываться в базе, а также как они будут взаимодействовать друг с другом. Это представление позволяет избежать избыточности данных, уменьшить возможность возникновения аномалий при обработке информации и гарантирует, что база данных будет нормализована, что в свою очередь способствует ее эффективному функционированию и высокой производительности. Разработка ER-диаграммы является неотъемлемой частью процесса проектирования, поскольку она помогает определить

структуру и логику базы данных до того, как она будет реализована в техническом плане.

На рисунке 5.1 представлена ER-диаграмма базы данных, отражающая ключевые сущности, их атрибуты и взаимосвязи, которые обеспечивают полноценное функционирование системы.

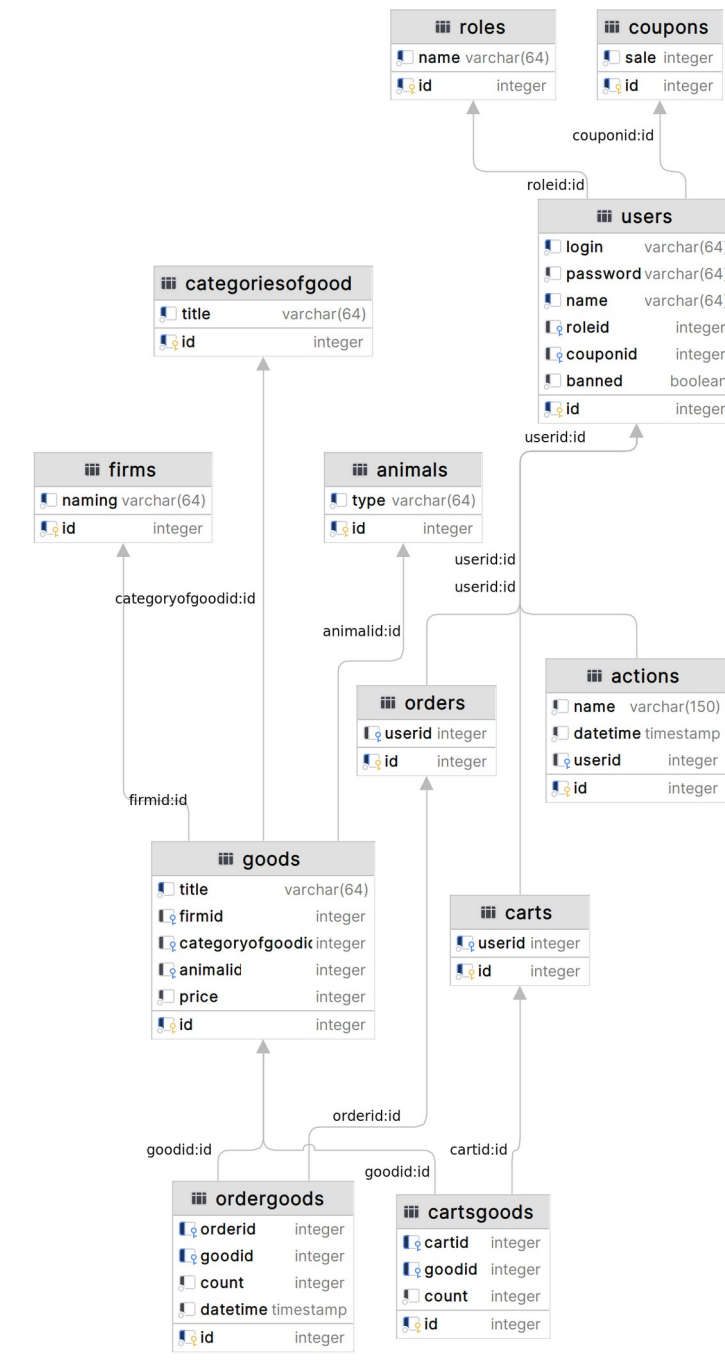


Рисунок 5.1 – ER-диаграмма базы данных

5.3 Оптимизация и целостность разработанной базы данных

Оптимизация структуры базы данных является ключевым этапом в проектировании информационной системы, направленным на повышение производительности, обеспечение целостности данных и минимизацию избыточности. Одним из способов оптимизации является использование триггеров, которые представляют собой специальные процедуры, автоматически выполняющиеся в ответ на изменения в данных. Триггеры могут использоваться для поддержания бизнес-логики, такой как проверка наличия определенных данных перед выполнением операции, или для автоматического выполнения дополнительных действий, например, создание записей в других таблицах.

Например, можно реализовать триггеры для предотвращения удаления или изменения критических данных, если они нарушают бизнес-правила, такие как отсутствие администратора в системе или попытка удаления данных, которые должны быть сохранены для целостности системы. Триггеры также могут использоваться для автоматического создания связанных записей в других таблицах, когда новые данные добавляются в базу, что исключает необходимость вручную контролировать такие зависимости.

Для оптимизации быстродействия базы данных важную роль играют индексы. Индексы создаются на столбцах, которые используются для поиска и фильтрации данных, таких как идентификаторы, внешние ключи или часто используемые атрибуты, например, названия товаров или электронные адреса пользователей. Индексы значительно ускоряют операции выборки, сортировки и соединения таблиц, однако важно, чтобы индексы использовались разумно, так как их избыточное применение может привести к ухудшению производительности при вставке, обновлении или удалении данных.

Еще одной важной частью оптимизации является нормализация базы данных. Нормализация представляет собой процесс реорганизации данных в таблицах с целью уменьшения избыточности и предотвращения аномалий, таких как дублирование данных или трудности при обновлении информации. Нормализация обеспечивает целостность данных и упрощает управление ими. Однако в некоторых случаях, например, при необходимости ускорения чтения данных, может быть разумным применение денормализации, когда часть данных избыточно сохраняется в таблицах для ускорения запросов.

Кроме того, для улучшения производительности базы данных могут быть использованы асинхронные операции и оптимизация запросов. Асинхронные запросы позволяют не блокировать выполнение других операций, что повышает скорость отклика системы. Оптимизация запросов включает использование более эффективных способов выборки данных, минимизацию количества соединений между таблицами и правильное использование агрегатных функций.

В итоге, оптимизация структуры базы данных направлена на обеспечение ее высокой производительности, минимизацию затрат на хранение данных и предотвращение возможных ошибок при их обработке. Эффективное использование триггеров, индексов, нормализации и других методов помогает добиться этих целей, создавая основу для надежной и быстродействующей системы.

5.4 Описание базы данных

На основе диаграммы базы данных, можно описать проектирование разработанной базы данных, включая сущности, связи и функциональное назначение каждой таблицы.

Сущность User представляет пользователей системы и включает такие атрибуты, как идентификатор пользователя, имя, адрес электронной почты, пароль и идентификатор роли. Связь с сущностью Role реализована по схеме "многие-к-одному", где каждому пользователю соответствует одна роль, например, администратор или клиент. Также сущность User связана с сущностями Cart ("один-к-одному", так как у одного пользователя может быть только одна корзина) и Order ("один-ко-многим", так как пользователь может иметь несколько заказов). Кроме того, связь с CouponUser ("многие-ко-многим") позволяет отслеживать использование купонов конкретными пользователями.

Сущность Role содержит перечень ролей с их уникальными идентификаторами и названиями. Она связана с User, чтобы каждому пользователю назначалась одна роль. Это связь типа "один-ко-многим", поскольку одна роль может быть назначена нескольким пользователям.

Сущность Cart представляет корзину покупателя и связана с User по схеме "один-к-одному", так как один пользователь может иметь только одну корзину. Также сущность Cart связана с сущностью CartGood по схеме "один-ко-многим", что позволяет добавлять в корзину несколько товаров.

Сущность CartGood связывает корзины и товары через их идентификаторы. Это промежуточная сущность, реализующая связь "многие-ко-многим" между Cart и Good. В ней хранятся данные о количестве конкретного товара в корзине.

Сущность Good представляет товары, доступные в магазине. Она связана с GoodsType ("многие-к-одному"), что позволяет классифицировать товары по типу, и с CategoryOfGood ("многие-к-одному"), что позволяет группировать товары по категориям. Также сущность Good связана с CartGood и OrderGood, что обеспечивает возможность добавления товаров в корзину и оформления заказов.

Сущность GoodsType определяет типы товаров, такие как "игрушки", "еда" или "аксессуары". Она связана с Good по схеме "один-ко-многим", так как один тип может включать множество товаров.

Сущность CategoryOfGood отвечает за категории товаров, такие как "для собак" или "для кошек". Связь с Good также реализована по схеме "один-ко-многим", что позволяет одной категории включать множество товаров.

Сущность Animal используется для описания животных, для которых предназначены товары. Она связана с Good по схеме "один-ко-многим", что позволяет определять, для какого животного предназначен каждый товар.

Сущность Order хранит информацию о заказах, сделанных пользователями. Она связана с User по схеме "многие-к-одному", так как один пользователь может иметь несколько заказов. Также она связана с OrderGood по схеме "один-ко-многим", чтобы включать информацию о заказанных товарах.

Сущность OrderGood связывает заказы и товары, реализуя связь "многие-ко-многим" между Order и Good. В ней хранятся данные о количестве каждого товара в заказе.

Сущность Coupon представляет скидочные купоны. Она связана с CouponUser по схеме "один-ко-многим", так как один купон может быть использован несколькими пользователями. Также она может быть связана с Order, чтобы фиксировать использование купонов в заказах.

Сущность CouponUser является промежуточной таблицей, реализующей связь "многие-ко-многим" между Coupon и User. Она фиксирует, какие купоны были использованы конкретным пользователем.

Сущность Logging хранит данные о действиях пользователей в системе. Она связана с User по схеме "многие-к-одному", так как один пользователь может выполнить множество действий, фиксируемых в логах.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы было проведено всестороннее исследование и разработка веб-приложения для зоомагазина. Рассмотрены ключевые аспекты работы таких приложений, начиная с их роли в оптимизации процессов управления товарными запасами, заказами и взаимодействием с клиентами, и заканчивая возможностями использования современных технологий для улучшения качества обслуживания. Особое внимание уделено анализу программных систем и технологий, используемых для создания эффективных веб-приложений, таких как базы данных, серверные технологии и интерфейсы взаимодействия с пользователем.

В работе был также проведен анализ потребностей зоомагазинов и их клиентов, выявлены основные задачи, решаемые с помощью веб-приложения, и предложены оптимальные решения для их реализации. В рамках разработки веб-приложения использовались такие технологии, как HTML, CSS, JavaScript для фронтенда, а также выбор базы данных и серверной части для хранения информации о товарах, заказах и клиентах.

Результатом работы стало создание полноценного веб-приложения, которое обеспечит удобство и доступность для клиентов, а также улучшит процесс управления магазином для владельцев и сотрудников. Приложение предлагает интуитивно понятный интерфейс, функции для поиска товаров, оформления заказов и взаимодействия с клиентами, что способствует улучшению качества обслуживания и повышению конкурентоспособности зоомагазина.

Таким образом, результатом выполнения курсовой работы стало веб-приложение для зоомагазина, которое успешно решает задачи автоматизации процессов торговли, упрощает взаимодействие с клиентами и позволяет бизнесу эффективно управлять ассортиментом и заказами. Это решение также способствует улучшению пользовательского опыта и повышению операционной эффективности бизнеса.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] Архитектура Zen: сколько поколений продержится главная технология AMD [Электронный ресурс]. – Режим доступа: <https://club.dns-shop.ru/blog/t-100-protssoryi/61416-arhitektura-zen-skolko-pokolenii-proderjitsya-glavnaya-tehnologi/> – Дата доступа: 01.10.2023.

[2] Поколения процессоров AMD Ryzen [Электронный ресурс]. – Режим доступа: <https://te4h.ru/pokoleniya-protssorov-amd-ryzen> – Дата доступа: 01.10.2023.

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг программного кода

create_tables.py

```
from config.project_config import Database

class TableCreator:

    @staticmethod
    async def create_roles_table():
        query = '''
            CREATE TABLE IF NOT EXISTS Roles (
                Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                Name VARCHAR (64) NOT NULL UNIQUE
            );
        '''
        await Database.execute(query)

    @staticmethod
    async def create_coupons_table():
        query = '''
            CREATE TABLE IF NOT EXISTS Coupons (
                Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                Sale INTEGER NOT NULL UNIQUE
            );
        '''
        await Database.execute(query)

    @staticmethod
    async def create_users_table():
        query = '''
            CREATE TABLE IF NOT EXISTS Users (
                Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                Login VARCHAR (64) NOT NULL UNIQUE,
                Password VARCHAR (64) NOT NULL,
                Name VARCHAR (64) NOT NULL UNIQUE,
                RoleId UUID REFERENCES Roles (Id) ON DELETE SET NULL,
                CouponId UUID REFERENCES Coupons (Id) ON DELETE SET
NULL,
                Banned BOOLEAN DEFAULT FALSE
            );
        '''
        await Database.execute(query)

    @staticmethod
    async def create_carts_table():
```

```

        query = '''
            CREATE TABLE IF NOT EXISTS Carts (
                Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                UserId UUID NOT NULL REFERENCES Users (Id) ON DELETE
CASCADE,
                Goods UUID[] DEFAULT '{}'
            );
        '''
        await Database.execute(query)

    @staticmethod
    async def create_firms_table():
        query = '''
            CREATE TABLE IF NOT EXISTS Firms (
                Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                Naming VARCHAR (64) NOT NULL UNIQUE
            );
        '''
        await Database.execute(query)

    @staticmethod
    async def create_animals_table():
        query = '''
            CREATE TABLE IF NOT EXISTS Animals (
                Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                Type VARCHAR (64) NOT NULL UNIQUE
            );
        '''
        await Database.execute(query)

    @staticmethod
    async def create_categories_of_good_table():
        query = '''
            CREATE TABLE IF NOT EXISTS CategoriesOfGood (
                Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                Title VARCHAR(64) NOT NULL UNIQUE
            );
        '''
        await Database.execute(query)

    @staticmethod
    async def create_goods_table():
        query = '''
            CREATE TABLE IF NOT EXISTS Goods (
                Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                Title VARCHAR (64) NOT NULL,
                FirmId UUID REFERENCES Firms (Id) ON DELETE CASCADE,

```

```

        CategoryOfGoodId UUID REFERENCES CategoriesOfGood (Id)
ON DELETE CASCADE,
        AnimalId UUID REFERENCES Animals (Id) ON DELETE CASCADE
    );
    '''
    await Database.execute(query)

@staticmethod
async def create_orders_table():
    query = '''
        CREATE TABLE IF NOT EXISTS Orders (
            Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
            UserId UUID REFERENCES Users (Id) ON DELETE SET NULL,
            Goods UUID[] DEFAULT ARRAY[]::UUID[]
        );
    '''
    await Database.execute(query)

@staticmethod
async def create_order_goods_table():
    query = '''
        CREATE TABLE IF NOT EXISTS OrderGoods (
            Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
            OrderId UUID REFERENCES Orders (Id) ON DELETE CASCADE,
            GoodId UUID REFERENCES Goods (Id) ON DELETE CASCADE,
            Count INTEGER NOT NULL DEFAULT 1,
            DateTime TIMESTAMP NOT NULL,
            UNIQUE (OrderId, GoodId)
        );
    '''
    await Database.execute(query)

@staticmethod
async def create_carts_goods_table():
    query = '''
        CREATE TABLE IF NOT EXISTS CartsGoods (
            Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
            CartId UUID REFERENCES Carts (Id) ON DELETE CASCADE,
            GoodId UUID REFERENCES Goods (Id) ON DELETE CASCADE,
            Count INTEGER NOT NULL DEFAULT 1,
            UNIQUE (CartId, GoodId)
        );
    '''
    await Database.execute(query)

@staticmethod
async def create_logging_table():
    query = '''
        CREATE TABLE IF NOT EXISTS Logs (

```

```

        Id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
        UserId UUID REFERENCES Users (Id) ON DELETE SET NULL,
        Role VARCHAR(255),
        Action VARCHAR(255),
        Result VARCHAR(255)
    );
    '''
    await Database.execute(query)

```

```

@staticmethod
async def create_all_tables():
    await TableCreator.create_roles_table()
    await TableCreator.create_coupons_table()
    await TableCreator.create_users_table()
    await TableCreator.create_carts_table()
    await TableCreator.create_firms_table()
    await TableCreator.create_animals_table()
    await TableCreator.create_categories_of_good_table()
    await TableCreator.create_goods_table()
    await TableCreator.create_orders_table()
    await TableCreator.create_order_goods_table()
    await TableCreator.create_carts_goods_table()
    await TableCreator.create_logging_table()

```

bd_queries.py

```

from typing import Optional, List, Dict
from uuid import UUID

from fastapi import HTTPException

from schemas.schemas import User, Good, UserProfile, Animal, Firm,
Coupon, Role, Category, GoodUpdate, GoodCreate, \
    LogCreate
from config.project_config import Database

class DatabaseQueries:

    @staticmethod
    async def get_user_coupons(login: str) -> Optional[List[dict]]:
        user = await Database.get_user_by_login(login)
        if not user:
            return None

        query = """
            SELECT u.Name AS User, c.Id AS CouponId, c.Sale
            FROM Coupons c
            JOIN Users u ON u.CouponId = c.Id
            WHERE u.Name = $1;
        """
        try:

```

```

        result = await Database.fetch(query, user['name'])
        return [dict(row) for row in result] if result else []
    except Exception as e:
        print(f"Error in get_user_coupons: {e}")
        return None

    @staticmethod
    async def get_goods_with_category() -> List[dict]:
        query = """
            SELECT g.Id, g.Title, c.Title AS Category
            FROM Goods g
            LEFT JOIN CategoriesOfGood c ON g.CategoryOfGoodId =
c.Id;

            """
        try:
            result = await Database.fetch(query)
            return [dict(row) for row in result]
        except Exception as e:
            print(f"Error in get_goods_with_category: {e}")
            return []

    @staticmethod
    async def get_good_by_animal(animal_get: str) ->
Optional[List[dict]]:
        if not animal_get:
            return None
        query = """
            SELECT g.Id, g.Title, a.Type AS Animal
            FROM Goods g
            LEFT JOIN Animals a ON g.AnimalId = a.Id
            WHERE a.Type = $1;

            """
        try:
            result = await Database.fetch(query, animal_get)
            return [dict(row) for row in result]
        except Exception as e:
            print(f"Error in get_good_by_animal: {e}")
            return None

    @staticmethod
    async def get_users() -> List[dict]:
        query = "SELECT u.Id, u.name, u.banned, u.login, u.roleid,
u.couponid FROM Users u;"
        try:
            result = await Database.fetch(query)
            return [dict(row) for row in result]
        except Exception as e:
            print(f"Error in get_users: {e}")

```

```

        return []

    @staticmethod
    async def get_user_profile(user_id: UUID) -> Optional[dict]:
        query = "SELECT * FROM Users WHERE Id = $1;"
        try:
            result = await Database.fetchrow(query, user_id)
            if result:
                return dict(result)
            return None
        except Exception as e:
            print(f"Error in get_user_profile: {e}")
            return None

    @staticmethod
    async def update_user(user_id: UUID, update_data: dict) -> bool:

        set_clause = []
        values = []
        index = 1

        if update_data.get("login"):
            set_clause.append(f"Login = ${index}")
            values.append(update_data["login"])
            index += 1
        if update_data.get("password"):
            set_clause.append(f"Password = ${index}")
            values.append(update_data["password"])
            index += 1
        if update_data.get("name"):
            set_clause.append(f"Name = ${index}")
            values.append(update_data["name"])
            index += 1

        if not set_clause:
            return False

        query = f"""
            UPDATE Users
            SET {'', ' '.join(set_clause)}
            WHERE Id = ${index}
            RETURNING Id;
        """
        values.append(user_id)

        try:
            result = await Database.fetchrow(query, *values)
            return result is not None
        except Exception as e:

```

```

        print(f"Error in update_user: {e}")
        return False

    @staticmethod
    async def delete_user(user_id: UUID) -> bool:

        cart_deleted = await DatabaseQueries.delete_cart(user_id)
        if not cart_deleted:
            print("Error: Cart could not be deleted.")

        query = """
            DELETE FROM Users WHERE Id = $1 RETURNING Id;
        """
        try:
            result = await Database.fetchrow(query, user_id)
            return result is not None
        except Exception as e:
            print(f"Error in delete_user: {e}")
            return False

    @staticmethod
    async def create_user(login: str, password: str, name: str) ->
Optional[UserProfile]:
        default_role_id = 'b3b9d771-010e-432d-9f06-f36e2269465f'
        query = """
            INSERT INTO Users (Login, Password, Name, RoleId,
CouponId, Banned)
            VALUES ($1, $2, $3, $4, NULL, FALSE) -- Значения по
умолчанию для RoleId, CouponId, и Banned
            RETURNING Id, Login, Name, RoleId, CouponId, Banned;
        """
        try:

            result = await Database.fetchrow(query, login, password,
name, default_role_id)
            if result:

                await DatabaseQueries.create_cart(result['id'])

                user = UserProfile(**result)
                return user
            return None
        except Exception as e:
            print(f"Error in create_user: {e}")
            return None

    @staticmethod
    async def get_all_animals():
        query = "SELECT * FROM Animals;"
        rows = await Database.fetch(query)
        return [dict(row) for row in rows]

```

```

@staticmethod
async def get_animal(animal_id: UUID) -> Optional[dict]:
    query = "SELECT * FROM Animals WHERE Id = $1;"
    try:
        result = await Database.fetchrow(query, animal_id)
        if result:
            return dict(result)
        return None
    except Exception as e:
        print(f"Error in get_animal: {e}")
        return None

@staticmethod
async def create_animal(animal_type: str) -> Optional[Animal]:
    query = """
        INSERT INTO Animals (Type)
        VALUES ($1)
        RETURNING Id, Type;
    """
    try:
        result = await Database.fetchrow(query, animal_type)

        if result:

            return Animal(id=result["id"], type=result["type"])
        return None
    except Exception as e:
        print(f"Error in create_animal: {e}")
        return None

@staticmethod
async def delete_animal(animal_id: UUID) -> bool:
    query = """
        DELETE FROM Animals WHERE Id = $1 RETURNING Id;
    """
    try:
        result = await Database.fetchrow(query, animal_id)
        return result is not None
    except Exception as e:
        print(f"Error in delete_animal: {e}")
        return False

@staticmethod
async def get_all_firms():
    query = "SELECT * FROM Firms;"
    rows = await Database.fetch(query)
    return [dict(row) for row in rows]

@staticmethod
async def get_firm(firm_id: UUID) -> Optional[dict]:
    query = "SELECT * FROM Firms WHERE Id = $1;"

```



```

    try:
        result = await Database.fetchrow(query, firm_id)
        if result:
            return dict(result)
        return None
    except Exception as e:
        print(f"Error in get_firm: {e}")
        return None

@staticmethod
async def create_firm(firm_name: str) -> Optional[dict]:
    query = """
        INSERT INTO Firms (Naming)
        VALUES ($1)
        RETURNING Id, Naming;
    """
    try:
        result = await Database.fetchrow(query, firm_name)

        if result:

            return Firm(id=result["id"], naming=result["naming"])
        return None
    except Exception as e:
        print(f"Error in create_firm: {e}")
        return None

@staticmethod
async def delete_firm(firm_id: UUID) -> bool:
    query = """
        DELETE FROM Firms WHERE Id = $1 RETURNING Id;
    """
    try:
        result = await Database.fetchrow(query, firm_id)
        return result is not None
    except Exception as e:
        print(f"Error in delete_firm: {e}")
        return False

@staticmethod
async def get_all_coupons():
    query = "SELECT * FROM Coupons;"
    rows = await Database.fetch(query)
    return [dict(row) for row in rows]

@staticmethod
async def get_coupon(coupon_id: UUID) -> Optional[dict]:
    query = "SELECT * FROM Coupons WHERE Id = $1;"
    try:
        result = await Database.fetchrow(query, coupon_id)
        if result:
            return dict(result)

```

```

        return None
    except Exception as e:
        print(f"Error in get_coupon: {e}")
        return None

@staticmethod
async def create_coupon(coupon_sale: int) -> Optional[dict]:
    query = """
        INSERT INTO Coupons (Sale)
        VALUES ($1)
        RETURNING Id, Sale;
    """
    try:
        result = await Database.fetchrow(query, coupon_sale)

        if result:

            return Coupon(id=result["id"], sale=result["sale"])
        return None
    except Exception as e:
        print(f"Error in create_coupon: {e}")
        return None

@staticmethod
async def delete_coupon(coupon_id: UUID) -> bool:
    query = """
        DELETE FROM Coupons WHERE Id = $1 RETURNING Id;
    """
    try:
        result = await Database.fetchrow(query, coupon_id)
        return result is not None
    except Exception as e:
        print(f"Error in delete_coupon: {e}")
        return False

@staticmethod
async def get_all_roles():
    query = "SELECT * FROM Roles;"
    rows = await Database.fetch(query)
    return [dict(row) for row in rows]

@staticmethod
async def get_role(role_id: UUID) -> Optional[dict]:
    query = "SELECT * FROM Roles WHERE Id = $1;"
    try:
        result = await Database.fetchrow(query, role_id)
        if result:
            return dict(result)
        return None
    except Exception as e:

```

```

        print(f"Error in get_role: {e}")
        return None

    @staticmethod
    async def create_role(role_name: str) -> Optional[dict]:
        query = """
            INSERT INTO Roles (Name)
            VALUES ($1)
            RETURNING Id, Name;
        """
        try:
            result = await Database.fetchrow(query, role_name)

            if result:

                return Role(id=result["id"], name=result["name"])
            return None
        except Exception as e:
            print(f"Error in create_role: {e}")
            return None

    @staticmethod
    async def delete_role(role_id: UUID) -> bool:
        query = """
            DELETE FROM Roles WHERE Id = $1 RETURNING Id;
        """
        try:
            result = await Database.fetchrow(query, role_id)
            return result is not None
        except Exception as e:
            print(f"Error in delete_role: {e}")
            return False

    @staticmethod
    async def get_all_categories():
        query = "SELECT * FROM Categoriesofgood;"
        rows = await Database.fetch(query)
        return [dict(row) for row in rows]

    @staticmethod
    async def get_category(category_id: UUID) -> Optional[dict]:
        query = "SELECT * FROM Categoriesofgood WHERE Id = $1;"
        try:
            result = await Database.fetchrow(query, category_id)
            if result:
                return dict(result)
            return None
        except Exception as e:
            print(f"Error in category get: {e}")
            return None

```

```

@staticmethod
async def create_category(category_title: str) -> Optional[dict]:
    query = """
        INSERT INTO Categoriesofgood (TITLE)
        VALUES ($1)
        RETURNING Id, Title;
    """
    try:
        result = await Database.fetchrow(query, category_title)

        if result:

            return Category(id=result["id"], title=result["title"])
        return None
    except Exception as e:
        print(f"Error in create_category: {e}")
        return None

@staticmethod
async def delete_category(category_id: UUID) -> bool:
    query = """
        DELETE FROM Categoriesofgood WHERE Id = $1 RETURNING Id;
    """
    try:
        result = await Database.fetchrow(query, category_id)
        return result is not None
    except Exception as e:
        print(f"Error in delete_role: {e}")
        return False

@staticmethod
async def create_cart(user_id: UUID) -> Optional[dict]:
    query = """
        INSERT INTO Carts (UserId, Goods)
        VALUES ($1, $2)
        RETURNING Id, UserId, Goods;
    """
    try:

        result = await Database.fetchrow(query, user_id, [])
        if result:
            return dict(result)
        return None
    except Exception as e:
        print(f"Error in create_cart: {e}")
        return None

@staticmethod
    async def add_good_to_cart(cart_id: UUID, good_id: UUID) ->
Optional[dict]:

```

```

        query_check_good = "SELECT 1 FROM Goods WHERE Id = $1 LIMIT 1;"
        good_exists = await Database.fetchval(query_check_good, good_id)

        if not good_exists:
            raise HTTPException(status_code=404, detail="Good not
found")

    query = """
        UPDATE Carts
        SET Goods = array_append(Goods, $1)
        WHERE Id = $2
        RETURNING Id, UserId, Goods;
    """
    try:
        result = await Database.fetchrow(query, good_id, cart_id)
        if result:
            return dict(result)
        return None
    except Exception as e:
        print(f"Error in add_good_to_cart: {e}")
        return None

    @staticmethod
    async def remove_good_from_cart(cart_id: UUID, good_id: UUID) ->
Optional[dict]:
        query = """
            UPDATE Carts
            SET Goods = array_remove(Goods, $1)
            WHERE Id = $2
            RETURNING Id, UserId, Goods;
        """
        try:
            result = await Database.fetchrow(query, good_id, cart_id)
            if result:
                return dict(result)
            return None
        except Exception as e:
            print(f"Error in remove_good_from_cart: {e}")
            return None

    @staticmethod
    async def get_cart_goods(cart_id: UUID) -> Optional[List[UUID]]:
        query = """
            SELECT Goods
            FROM Carts
            WHERE Id = $1;
        """
        try:
            result = await Database.fetchval(query, cart_id)
            if result is not None:
                return result
            return None

```

```

except Exception as e:
    print(f"Error in get_cart_goods: {e}")
    return None

@staticmethod
async def get_cart_by_user_id(user_id: UUID) -> Optional[dict]:
    query = "SELECT * FROM Carts WHERE UserId = $1;"
    try:
        result = await Database.fetchrow(query, user_id)
        if result:
            return dict(result)
        return None
    except Exception as e:
        print(f"Error in get_cart_by_user_id: {e}")
        return None

@staticmethod
async def delete_cart(cart_id: UUID) -> bool:
    query = """
        DELETE FROM Carts WHERE Id = $1 RETURNING Id;
    """
    try:
        result = await Database.fetchrow(query, cart_id)
        return result is not None
    except Exception as e:
        print(f"Error in delete_cart: {e}")
        return False

@staticmethod
async def create_order(user_id: UUID) -> Optional[dict]:
    query = """
        INSERT INTO Orders (UserId)
        VALUES ($1)
        RETURNING Id, UserId, Goods;
    """
    try:
        result = await Database.fetchrow(query, user_id)
        if result:
            return dict(result)
        return None
    except Exception as e:
        print(f"Error in create_order: {e}")
        return None

@staticmethod
async def get_all_orders() -> List[dict]:
    query = """
        SELECT Id, UserId, Goods
        FROM Orders;
    """
    try:
        results = await Database.fetch(query)
        return [dict(row) for row in results]

```

```

        except Exception as e:
            print(f"Error in get_all_orders: {e}")
            return []

    @staticmethod
    async def get_order_by_id1(order_id: UUID):
        query = "SELECT * FROM Orders WHERE Id = $1"
        result = await Database.fetchrow(query, order_id)
        return result

    @staticmethod
    async def delete_order(order_id: UUID):
        query = "DELETE FROM Orders WHERE Id = $1 RETURNING Id"
        try:
            result = await Database.fetchval(query, order_id)
            return result is not None
        except Exception as e:
            print(f"Error deleting order {order_id}: {e}")
            return False

    @staticmethod
    async def add_good_to_order(user_id: UUID, good_id: UUID) ->
Optional[dict]:

        good_exists = await DatabaseQueries.check_good_exists(good_id)
        if not good_exists:
            raise HTTPException(status_code=400, detail="Good not
found")

        query_get_cart = """
        SELECT Id
        FROM Carts
        WHERE UserId = $1;
        """
        cart_id = await Database.fetchval(query_get_cart, user_id)
        if cart_id:
            await DatabaseQueries.remove_good_from_cart(cart_id,
good_id)

        query_get_order = """
        SELECT Id
        FROM Orders
        WHERE UserId = $1 AND cardinality(Goods) > 0;
        """
        existing_order_id = await Database.fetchval(query_get_order,
user_id)

        if not existing_order_id:
            create_order_query = """

```

```

        INSERT INTO Orders (UserId, Goods)
        VALUES ($1, ARRAY[$2]::UUID[])
        RETURNING Id, UserId, Goods;
    """
    try:
        result = await Database.fetchrow(create_order_query,
user_id, good_id)
        if result:
            return dict(result)
        return None
    except Exception as e:
        print(f"Error in create_order within add_good_to_order:
{e}")
        raise HTTPException(status_code=500, detail="Failed to
create order")
    else:

        add_good_query = """
        UPDATE Orders
        SET Goods = array_append(Goods, $1)
        WHERE Id = $2
        RETURNING Id, UserId, Goods;
        """
        try:
            result = await Database.fetchrow(add_good_query,
good_id, existing_order_id)
            if result:
                return dict(result)
            return None
        except Exception as e:
            print(f"Error in add_good_to_order: {e}")
            raise HTTPException(status_code=500, detail="Failed to
add good to order")

    @staticmethod
    async def check_good_exists(good_id: UUID) -> bool:
        query = """
        SELECT EXISTS (
            SELECT 1
            FROM Goods
            WHERE Id = $1
        );
        """
        try:
            result = await Database.fetchval(query, good_id)
            return result
        except Exception as e:
            print(f"Error in check_good_exists: {e}")
            raise HTTPException(status_code=500, detail="Internal server
error during good existence check")

    @staticmethod

```



```

        async def remove_good_from_order(order_id: UUID, good_id: UUID) ->
Optional[dict]:
            query = """
                UPDATE Orders
                SET Goods = array_remove(Goods, $1)
                WHERE Id = $2
                RETURNING Id, UserId, Goods;
            """
            try:
                result = await Database.fetchrow(query, good_id, order_id)

                if result:

                    if not result["goods"]:
                        delete_order_query = """
                            DELETE FROM Orders
                            WHERE Id = $1;
                        """
                        await Database.execute(delete_order_query, order_id)
                        return {"detail": "Order deleted because it was
empty"}

                    return dict(result)
                return None
            except Exception as e:
                print(f"Error in remove_good_from_order: {e}")
                raise HTTPException(status_code=500, detail="Failed to
remove good from order")

    @staticmethod
    async def get_order_goods(order_id: UUID) -> Optional[List[UUID]]:
        query = """
            SELECT Goods
            FROM Orders
            WHERE Id = $1;
        """
        try:
            print(f"Result from database for order {order_id}:
{Database.fetchval(query, order_id)}")
            result = await Database.fetchval(query, order_id)
            if result is not None:
                return result
            return None
        except Exception as e:
            print(f"Error in get_order_goods: {e}")
            return None

    @staticmethod
    async def get_orders_by_user_id(user_id: UUID) -> List[dict]:
        query = """
            SELECT Id, UserId, Goods
            FROM Orders
            WHERE UserId = $1;
        """

```

```

    try:
        results = await Database.fetch(query, user_id)
        print(f"result:{results}")
        return [dict(row) for row in results]
    except Exception as e:
        print(f"Error in get_orders_by_user_id: {e}")
        return []

@staticmethod
async def remove_order(order_id: UUID) -> bool:
    query = """
        DELETE FROM Orders
        WHERE Id = $1;
    """
    try:
        result = await Database.execute(query, order_id)
        return result == "DELETE 1"
    except Exception as e:
        print(f"Error in remove_order: {e}")
        return False

@staticmethod
async def get_all_goods() -> List[Good]:
    query = """
        SELECT g.Id, g.Title, g.FirmId, g.CategoryOfGoodId,
g.AnimalId, g.Price, g.ImageURL
        FROM Goods g;
    """
    try:
        results = await Database.fetch(query)
        return [Good(**result) for result in results]
    except Exception as e:
        print(f"Error in get_all_goods: {e}")
        return []

@staticmethod
async def get_good_by_id(good_id: UUID) -> Optional[Good]:
    query = """
        SELECT g.Id, g.Title, g.FirmId, g.CategoryOfGoodId,
g.AnimalId, g.Price, g.ImageURL
        FROM Goods g WHERE g.Id = $1;
    """
    try:
        result = await Database.fetchrow(query, good_id)
        if result:
            return Good(**result)
        return None
    except Exception as e:
        print(f"Error in get_good_by_id: {e}")
        return None

```

```

    @staticmethod
    async def delete_good(good_id: UUID) -> bool:
        query = "DELETE FROM Goods WHERE Id = $1 RETURNING Id;"
        try:
            result = await Database.fetchrow(query, good_id)
            if result:
                return True
            return False
        except Exception as e:
            print(f"Error in delete_good: {e}")
            return False

    @staticmethod
    async def update_good(good_id: UUID, good_update: GoodUpdate) ->
Optional[Good]:

        query_get_current = "SELECT * FROM Goods WHERE Id = $1;"
        current_good = await Database.fetchrow(query_get_current,
good_id)

        if not current_good:
            raise HTTPException(status_code=404, detail="Good not
found")

        # Извлекаем текущие данные
        current_good = dict(current_good)

        # Если новые значения переданы, проверяем их
        if good_update.firmId:
            firm_exists = await
DatabaseQueries.check_firm_exists(good_update.firmId)
            if not firm_exists:
                raise HTTPException(status_code=400, detail="Firm not
found")

            if good_update.categoryOfGoodId:
                category_exists = await
DatabaseQueries.check_category_exists(good_update.categoryOfGoodId)
                if not category_exists:
                    raise HTTPException(status_code=400, detail="Category
not found")

            if good_update.animalId:
                animal_exists = await
DatabaseQueries.check_animal_exists(good_update.animalId)
                if not animal_exists:
                    raise HTTPException(status_code=400, detail="Animal not
found")

        # Проверка на дублирование названия (если название обновляется)
        if good_update.title and good_update.title !=
current_good["title"]:
            existing_good_query = """
            SELECT 1
            FROM Goods
            WHERE Title = $1 AND Id != $2;

```

```

        """
        existing_good = await Database.fetchval(existing_good_query,
good_update.title, good_id)
        if existing_good:
            raise HTTPException(status_code=400, detail="A good with
this title already exists")

        # Обновляем только переданные значения, остальные оставляем как
есть
        updated_values = {
            "title": good_update.title or current_good["title"],
            "firmId": good_update.firmId or current_good["firmid"],
            "categoryOfGoodId": good_update.categoryOfGoodId or
current_good["categoryofgoodid"],
            "animalId": good_update.animalId or
current_good["animalid"],
            "price": good_update.price or current_good["price"],
            "imageUrl": good_update.imageUrl or
current_good["imageurl"],
        }

        # Формируем запрос на обновление
        query_update = """
        UPDATE Goods
        SET Title = $1, FirmId = $2, CategoryOfGoodId = $3, AnimalId
= $4, Price = $5, ImageURL = $6
        WHERE Id = $7
        RETURNING Id, Title, FirmId, CategoryOfGoodId, AnimalId,
Price, ImageURL;
        """
        try:
            result = await Database.fetchrow(
                query_update,
                updated_values["title"],
                updated_values["firmId"],
                updated_values["categoryOfGoodId"],
                updated_values["animalId"],
                updated_values["price"],
                updated_values["imageUrl"],
                good_id,
            )
            if result:
                return Good(**result)
            return None
        except Exception as e:
            print(f"Error in update_good: {e}")
            raise HTTPException(status_code=500, detail="Internal server
error")

    @staticmethod
    async def check_firm_exists(firm_id: UUID) -> bool:
        query = "SELECT 1 FROM Firms WHERE Id = $1;"
        try:
            result = await Database.fetchrow(query, firm_id)

```

```

        return result is not None
    except Exception as e:
        print(f"Error in check_firm_exists: {e}")
        return False

    @staticmethod
    async def check_category_exists(category_id: UUID) -> bool:
        query = "SELECT 1 FROM CategoriesOfGood WHERE Id = $1;"
        try:
            result = await Database.fetchrow(query, category_id)
            return result is not None
        except Exception as e:
            print(f"Error in check_category_exists: {e}")
            return False

    @staticmethod
    async def check_animal_exists(animal_id: UUID) -> bool:
        query = "SELECT 1 FROM Animals WHERE Id = $1;"
        try:
            result = await Database.fetchrow(query, animal_id)
            return result is not None
        except Exception as e:
            print(f"Error in check_animal_exists: {e}")
            return False

    @staticmethod
    async def add_good(good: GoodCreate) -> Optional[Good]:
        # Проверка на существование фирм, категорий и животных
        firm_exists = await
DatabaseQueries.check_firm_exists(good.firmId)
        category_exists = await
DatabaseQueries.check_category_exists(good.categoryOfGoodId)
        animal_exists = await
DatabaseQueries.check_animal_exists(good.animalId)

        if not firm_exists:
            raise HTTPException(status_code=400, detail="Firm not
found")
        if not category_exists:
            raise HTTPException(status_code=400, detail="Category not
found")
        if not animal_exists:
            raise HTTPException(status_code=400, detail="Animal not
found")

        # Проверка на наличие товара с таким же названием
        existing_good_query = """
        SELECT 1
        FROM Goods
        WHERE Title = $1
        """
        existing_good = await Database.fetchval(existing_good_query,
good.title)

```

```

        if existing_good:
            raise HTTPException(status_code=400, detail="A good with
this title already exists")

        # Если все проверки пройдены, добавляем товар
        query = """
            INSERT INTO Goods (Title, FirmId, CategoryOfGoodId,
AnimalId, Price, ImageURL)
            VALUES ($1, $2, $3, $4, $5, $6)
            RETURNING Id, Title, FirmId, CategoryOfGoodId, AnimalId,
Price, ImageURL;
        """
        try:
            result = await Database.fetchrow(
                query, good.title, good.firmId, good.categoryOfGoodId,
good.animalId, good.price, good.imageURL
            )
            if result:
                return Good(**result)
            return None
        except Exception as e:
            print(f"Error in add_good: {e}")
            return None

    @staticmethod
    async def get_logging() -> Optional[list]:
        query = """
            SELECT * FROM Logs
            ORDER BY Timestamp DESC;
        """
        try:
            result = await Database.fetch(query)
            return [dict(record) for record in result]
        except Exception as e:
            print(f"Error in get_logging: {e}")
            return None

    @staticmethod
    async def get_logging_by_user(user_id: UUID) -> Optional[list]:
        query = """
            SELECT * FROM Logs
            WHERE UserId = $1
            ORDER BY Timestamp DESC;
        """
        try:
            result = await Database.fetch(query, user_id)
            return [dict(record) for record in result]
        except Exception as e:
            print(f"Error in get_logging_by_user: {e}")
            return None

    @staticmethod
    async def add_log(log: LogCreate) -> Optional[dict]:
        query = """

```

```

        INSERT INTO Logs (UserId, Role, Action, Result)
        VALUES ($1, $2, $3, $4)
        RETURNING Id, UserId, Role, Action, Timestamp, Result;
"""
try:
    result = await Database.fetchrow(
        query,
        log.userid,
        log.role,
        log.action,
        log.result
    )
    return dict(result) if result else None
except Exception as e:
    print(f"Error in add_log: {e}")
    return None

async def get_user_by_login(username: str):
    query = """
        SELECT
            Id,
            Login,
            Password,
            Name,
            RoleId,
            Banned
        FROM Users
        WHERE Login = $1
    """
    try:
        user = await Database.fetchrow(query, username)
        if not user:
            return None
        # Преобразуем результат в словарь для работы с Python-кодом
        return {
            "id": user["id"],
            "login": user["login"],
            "password": user["password"],
            "name": user["name"],
            "roleId": user["roleid"],
            "banned": user["banned"],
        }
    except Exception as e:
        print(f"Error in get_user_by_login: {e}")
        raise HTTPException(status_code=500, detail="Failed to fetch
user from database")

    @staticmethod
    async def get_role_by_id(role_id: UUID) -> Optional[dict]:
        query = "SELECT * FROM Roles WHERE Id = $1;"
        try:
            # Выполняем запрос к базе данных

```

```

        result = await Database.fetchrow(query, role_id)
        print(result["name"])
        if result:
            # Возвращаем результат в виде словаря
            return result["name"]
        return None
    except Exception as e:
        # Логгируем ошибку, если она возникла
        print(f"Error in get_role: {e}")
        return None

    @staticmethod
    async def add_log_from_dict(log: Dict):
        query = """
            INSERT INTO Logs (UserId, Role, Action, Result, Timestamp)
            VALUES ($1, $2, $3, $4, $5)
        """
        try:
            await Database.execute(
                query,
                log.get("user_id"),
                log.get("role"),
                log.get("action"),
                log.get("result"),
                log.get("timestamp")
            )
        except Exception as e:
            print(f"Error in add_log_from_dict: {e}")

    @staticmethod
    async def get_user_by_id(user_id: UUID):
        query = "SELECT * FROM Users WHERE Id = $1"
        return await Database.fetchrow(query, user_id)

    @staticmethod
    async def update_user_ban_status(user_id: UUID, banned: bool):
        query = "UPDATE Users SET Banned = $1 WHERE Id = $2 RETURNING
Id"

        try:
            result = await Database.fetchval(query, banned, user_id)
            return result is not None
        except Exception as e:
            print(f"Error updating ban status for user {user_id}: {e}")
            return False

```

schemas.py

```

from datetime import datetime
from uuid import UUID

from pydantic import BaseModel, Field
from typing import Optional, List

from pydantic import BaseModel

```



```

class LogCreate(BaseModel):
    userid: Optional[UUID] = None
    role: Optional[str] = None
    action: str
    result: Optional[str] = None

class LogResponse(LogCreate):
    id: UUID
    timestamp: datetime

class OrderItemUpdate(BaseModel):
    cart_id: UUID
    good_id: UUID

class Order(BaseModel):
    id: UUID
    userId: UUID
    goods: List[UUID] = []

    class Config:
        from_attributes = True
        alias_generator = lambda name: name.lower()

class GoodBase(BaseModel):
    title: str
    firmId: UUID
    categoryOfGoodId: UUID
    animalId: UUID
    price: float
    imageURL: str # Новое поле для хранения URL изображения

class GoodCreate(GoodBase):
    class Config:
        from_attributes = True
        alias_generator = lambda name: name.lower()

class GoodUpdate(BaseModel):
    title: Optional[str] = None
    firmId: Optional[UUID] = None
    categoryOfGoodId: Optional[UUID] = None
    animalId: Optional[UUID] = None
    price: Optional[float] = None
    imageURL: Optional[str] = None

class Good(GoodBase):
    id: UUID

    class Config:
        from_attributes = True

```

```

        alias_generator = lambda name: name.lower()

class CartItemUpdate(BaseModel):
    cart_id: UUID
    good_id: UUID

class Cart(BaseModel):
    id: UUID
    userId: UUID
    goods: List[UUID] = []

    class Config:
        from_attributes = True
        alias_generator = lambda name: name.lower()

class Category(BaseModel):
    id: UUID
    title: str

    class Config:
        from_attributes = True

class CategoryCreate(BaseModel):
    title: str

    class Config:
        from_attributes = True

class Role(BaseModel):
    id: UUID
    name: str

    class Config:
        from_attributes = True

class RoleCreate(BaseModel):
    name: str

    class Config:
        from_attributes = True

class Coupon(BaseModel):
    id: UUID
    sale: int

    class Config:
        from_attributes = True

class CouponCreate(BaseModel):
    sale: int

    class Config:
        from_attributes = True

```

```

class Firm(BaseModel):
    id: UUID
    naming: str

    class Config:
        from_attributes = True

class FirmCreate(BaseModel):
    naming: str

    class Config:
        from_attributes = True

class Animal(BaseModel):
    id: UUID = Field(..., alias="id")
    type: str = Field(..., alias="type")

class AnimalCreate(BaseModel):
    type: str

class UserCreate(BaseModel):
    login: str
    password: str
    name: str

    class Config:
        from_attributes = True

class UserUpdate(BaseModel):
    login: Optional[str] = Field(None, max_length=64)
    password: Optional[str] = Field(None, max_length=64)
    name: Optional[str] = Field(None, max_length=64)

    class Config:
        from_attributes = True

class UserProfile(BaseModel):
    id: UUID
    login: str
    name: str
    roleId: UUID
    couponId: Optional[UUID] = None
    banned: bool

    class Config:
        from_attributes = True
        alias_generator = lambda s: s.lower()

class User(BaseModel):
    id: UUID
    name: str

```

```

    login:str
    roleid : UUID
    couponid: Optional[UUID] = None
    banned: bool

    class Config:
        from_attributes = True

class LoginRequest(BaseModel):
    username: str
    password: str

app.js
const API_URL = 'http://127.0.0.1:8000';

// Проверка токена и управление кнопкой логина/логаута
document.addEventListener('DOMContentLoaded', () => {
    const authButton = document.getElementById('authButton');
    const token = localStorage.getItem('accessToken');

    if (token) {
        // Пользователь залогинен, показываем "Log Out"
        authButton.textContent = 'Log Out';
        authButton.href = '#';
        authButton.addEventListener('click', async () => {
            try {
                await logout();
            } catch (error) {
                console.error('Logout failed', error);
            }
        });

        // Проверяем роль пользователя и подгружаем интерфейс
        fetchUserProfile();
    } else {
        authButton.textContent = 'Log In';
        authButton.href = 'html/auth.html';
    }

    fetchGoods();
});

// Logout функция
async function logout() {
    const token = localStorage.getItem('accessToken');
    const response = await fetch(`${API_URL}/auth/logout`, {
        method: 'POST',
        headers: {
            'Authorization': `Bearer ${token}`
        }
    });
});

```

```

    if (response.ok) {
        localStorage.removeItem('accessToken');
        console.log('Logged out');
        window.location.reload();
    } else {
        console.log('Logout failed');
    }
}

// Проверяем профиль пользователя и роль
async function fetchUserProfile() {
    const token = localStorage.getItem('accessToken');
    if (!token) return;

    try {
        const response = await fetch(`${API_URL}/auth/me`, {
            headers: {
                'Authorization': `Bearer ${token}`
            }
        });

        if (response.ok) {
            const userData = await response.json();

            // Проверяем роль пользователя
            if (userData.roleid) {
                const roleResponse = await fetch(`${API_URL}/roles/${
                    userData.roleid}`, {
                    headers: { 'Authorization': `Bearer ${token}` }
                });

                if (roleResponse.ok) {
                    const roleData = await roleResponse.json();
                    if (roleData.name === 'admin') {
                        displayAdminMenu();
                    }
                }
            }
        }
    } catch (error) {
        console.error('Failed to fetch user profile', error);
    }
}

// Функция для отображения административных кнопок
function displayAdminMenu() {
    const ordersButton = document.getElementById('ordersButton');
    const logsButton = document.getElementById('logsButton');
    const usersButton = document.getElementById('usersButton');
    const dataButton = document.createElement('li'); // Создаем кнопку
    "Data"

    // Настройка кнопки Data
    dataButton.className = 'nav-item';

```

```

        dataButton.innerHTML = `
            <a href="../html/data.html" class="nav-link btn btn-outline-
light ml-3">Data</a>
        `;

        const nav = document.querySelector('.nav');
        nav.appendChild(dataButton);

        // Показ админских кнопок
        ordersButton.style.display = 'block';
        logsButton.style.display = 'block';
        usersButton.style.display = 'block';
        dataButton.style.display = 'block';
    }

    // Получаем все товары
    async function fetchGoods() {
        try {
            const response = await fetch(`${API_URL}/goods/`);
            const goods = await response.json();
            displayGoods(goods);

            // Получаем данные для выпадающих списков
            fetchAnimals();
            fetchFirms();
            fetchCategories();
        } catch (error) {
            console.error('Error loading goods:', error);
            document.getElementById('content').innerHTML = '<p>Failed to
load goods. Please try again later.</p>';
        }
    }

    // Получаем животных
    async function fetchAnimals() {
        try {
            const response = await fetch(`${API_URL}/animals/`);
            const animals = await response.json();
            populateDropdown('animalsDropdown', animals, 'animalid',
'type');
        } catch (error) {
            console.error('Error fetching animals:', error);
        }
    }

    // Получаем фирмы
    async function fetchFirms() {
        try {
            const response = await fetch(`${API_URL}/firms/`);
            const firms = await response.json();
            populateDropdown('firmsDropdown', firms, 'firmid', 'naming');
        } catch (error) {

```

```

        console.error('Error fetching firms:', error);
    }
}

// Получаем категории
async function fetchCategories() {
    try {
        const response = await fetch(`${API_URL}/categories/`);
        const categories = await response.json();
        populateDropdown('categoriesDropdown', categories,
'categoryofgoodid', 'title');
    } catch (error) {
        console.error('Error fetching categories:', error);
    }
}

// Отображение товаров
function displayGoods(goods) {
    const content = document.getElementById('content');
    content.innerHTML = '';

    goods.forEach(good => {
        const div = document.createElement('div');
        div.className = 'col-md-4 mb-4';

        div.innerHTML = `
            <div class="card" style="cursor: pointer;">
                
                <div class="card-body text-center">
                    <h5 class="card-title">${good.title}</h5>
                    <p class="card-text">${good.price}</p>
                    <button class="btn btn-primary add-to-cart"
style="background-color: #8395a6; border: #8395a6" data-id="${good.id}">Add
to Cart</button>
                </div>
            </div>
        `;

        // Добавляем обработчик клика на карточку
        div.querySelector('.card').addEventListener('click', () => {
            window.location.href = `../html/product.html?id=${good.id}`;
        });

        // Добавляем карточку на страницу
        content.appendChild(div);
    });

    // Обработчики для кнопок "Add to Cart"
    document.querySelectorAll('.add-to-cart').forEach(button => {
        button.addEventListener('click', async (e) => {
            e.stopPropagation(); // Останавливаем событие клика на
карточке

            const productId = button.dataset.id;

```

```

const token = localStorage.getItem('accessToken');

if (!token) {
  alert('You must log in first.');
```

<../html/auth.html>

```

  return;
}

let cartId = localStorage.getItem('cartid');

// Если корзина отсутствует, запросим её
if (!cartId) {
  try {
    const userResponse = await
fetch(`${API_URL}/auth/me`, {
  headers: { 'Authorization': `Bearer ${token}` }
});

    if (userResponse.ok) {
      const userData = await userResponse.json();
      const cartResponse = await
fetch(`${API_URL}/carts/${userData.id}`, {
        headers: { 'Authorization': `Bearer $
{token}` }

      });

      if (cartResponse.ok) {
        const cartData = await cartResponse.json();
        cartId = cartData.id;
        localStorage.setItem('cartid', cartId);
      } else {
        console.log('Could not fetch cart.');
```

Could not fetch cart.

```

        return;
      }
    } else {
      console.log('Could not fetch user profile.');
```

Could not fetch user profile.

```

      return;
    }
  } catch (error) {
    console.error('Error fetching cart ID:', error);
    console.log('An unexpected error occurred');
```

An unexpected error occurred

```

    return;
  }
}

// Отправляем товар в корзину
try {
  const response = await
fetch(`${API_URL}/carts/add_good`, {
    method: 'POST',
    headers: {
      'Authorization': `Bearer ${token}`,
      'Content-Type': 'application/json',
    },
  },

```



```

        body: JSON.stringify({ cart_id: cartId, good_id:
productId })),
    });

    if (response.ok) {
        console.log('Product added to cart successfully!');
    } else {
        const errorData = await response.json();
        console.log(`Failed to add product to cart: $
{errorData.detail}`);
    }
    } catch (error) {
        console.error('Error adding product to cart:', error);
        console.log('An unexpected error occurred');
    }
    });
    });
}

// Фильтрация товаров по выбранному критерию
async function filterGoods(filterType, filterValue) {
    try {
        const response = await fetch(`${API_URL}/goods/`);
        const goods = await response.json();
        const filteredGoods = goods.filter(good => good[filterType] ===
filterValue);
        displayGoods(filteredGoods);
    } catch (error) {
        console.error('Filter error:', error);
    }
}

// Заполняем выпадающее меню
function populateDropdown(dropdownId, items, filterType, displayField) {
    const dropdown = document.getElementById(dropdownId);
    dropdown.innerHTML = '';

    items.forEach(item => {
        const a = document.createElement('a');
        a.className = 'dropdown-item';
        a.href = '#';
        a.textContent = item[displayField] || "Unknown";
        a.addEventListener('click', () => filterGoods(filterType,
item.id));
        dropdown.appendChild(a);
    });
}

```

utils.js

```
const API_URL = 'http://127.0.0.1:8000';
```

```
// Получение токена из localStorage
```

```

function getToken() {
    return localStorage.getItem('accessToken');
}

// Проверяем наличие токена
function isLoggedIn() {
    const token = getToken();
    return !!token;
}

// Загрузка всех заказов при загрузке страницы
document.addEventListener('DOMContentLoaded', async () => {
    if (!isLoggedIn()) {
        console.log('You must log in to view orders');
        window.location.href = 'html/auth.html'; // Перенаправляем на
страницу логина
    } else {
        await fetchOrders();
    }
});

// Функция для загрузки всех заказов
async function fetchOrders() {
    try {
        const response = await fetch(`${API_URL}/orders`, {
            headers: {
                'Authorization': `Bearer ${getToken()}`
            }
        });

        if (response.ok) {
            const orders = await response.json();
            populateOrdersTable(orders);
        } else {
            console.log('Failed to load orders');
        }
    } catch (error) {
        console.error('Error while fetching orders', error);
        console.log('An error occurred while fetching orders');
    }
}

// Отображаем заказы в таблице
function populateOrdersTable(orders) {
    const tableBody = document.querySelector("#ordersTable tbody");
    tableBody.innerHTML = ''; // Очищаем таблицу перед новой загрузкой

    if (orders.length === 0) {
        tableBody.innerHTML = `
            <tr>
                <td colspan="4" class="text-center">No orders found</td>
            </tr>
        `;
        return;
    }
}

```

```

    }

    orders.forEach(order => {
      const row = document.createElement('tr');
      row.innerHTML = `
        <td>${order.id}</td>
        <td>${order.userid}</td>
        <td>${order.goods.join(', ')}</td>
        <td>
          <button class="btn btn-danger btn-sm"
onclick="deleteOrder('${order.id}')">Remove</button>
        </td>
      `;
      tableBody.appendChild(row);
    });
  }

  // Функция для удаления заказа
  async function deleteOrder(orderId) {
    if (!confirm('Are you sure you want to delete this order?')) {
      return;
    }

    try {
      const response = await fetch(`${API_URL}/orders/${orderId}`, {
        method: 'DELETE',
        headers: {
          'Authorization': `Bearer ${getToken()}`
        }
      });

      if (response.ok) {
        console.log('Order deleted successfully');
        await fetchOrders(); // Перезагружаем список заказов
      } else {
        const error = await response.json();
        console.log(`Failed to delete order: ${error.detail ||
'Unknown error'}`);
      }
    } catch (error) {
      console.error('Error while deleting order', error);
      console.log('An error occurred while deleting the order');
    }
  }
}

```

ПРИЛОЖЕНИЕ Б
(обязательное)
Конечная схема базы данных

ПРИЛОЖЕНИЕ В
(обязательное)
Ведомость курсового проекта