Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования

ИРКУТСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт информационных технологий и анализа данных
наименование института

Допускаю к защите
Руководитель ООП
_____ А.Ф. Аношко
подпись И.О. фамилия

Разработка программного средства для проектирования модели данных
наименование темы

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к выпускной квалификационной работе бакалавра
Программа бакалавриата
Вычислительные машины, комплексы, системы и сети
наименование программы

по направлению подготовки
09.03.01 «Информатика и вычислительная техника»
Код и наименование направления подготовки

0. 079.00.00 ПЗ
обозначение документа

Разработал студент группы ЭВМб-20-1 _____ Е.В. Прохоров
подпись И.О. Фамилия

Руководитель _____ А.С. Дорофеев
подпись И.О. Фамилия

Нормоконтроль _____ А.С. Дорофеев
подпись И.О. Фамилия

Иркутск 2024 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования

ИРКУТСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт информационных технологий и анализа данных
наименование института

УТВЕРЖДАЮ
Директор института ИТ и АД
_____А.С. Говорков
«_27_» _апреля_ 2024 г.

ЗАДАНИЕ

на выпускную квалификационную работу студенту Прохорову Евгению Викторовичу
группы ЭВМб-20-1

1 Тема работы: Разработка программного средства для проектирования модели данных

Утверждена приказом по университету от _____ № _____

2 Срок представления студентом законченной работы в ГЭК _____

3 Исходные данные
3.1 СТО 005-2020 «Система менеджмента качества. Учебно-методическая деятельность. Оформление курсовых проектов (работ) и выпускных квалификационных работ технических направлений подготовки и специальностей»
3.2 Материалы преддипломной практики

4 Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов):
4.1 Анализ языка SQL
4.2 Выбор стека технологий
4.3 Проектирование интерфейса
4.4 Проектирование базы данных
4.5 Создание программного средства
4.6 Тестирование

5 Перечень графического материала (с указанием обязательных чертежей)
5.1 Презентационные материалы

Календарный план

Разделы	Месяцы и недели												
	апрель				май					июнь			
Введение			+										
1. Анализ языка SQL				+									
2. Выбор стека технологий					+								
3. Проектирование интерфейса						+	+						
4. Проектирование базы данных						+	+						
5. Создание программного средства							+	+	+				
6. Тестирование										+			
Заключение										+			
Оформление пояснительной записки			+	+	+	+	+	+	+	+	+		
Подготовка к защите ВКР										+	+	+	+

Дата выдачи задания « 22 » апреля 2024 г.

Руководитель выпускной работы бакалавра _____ А.С. Дорофеев
подпись И.О. Фамилия

Руководитель ООП _____ А.Ф. Аношко
подпись И.О. Фамилия

Задание принял к исполнению студент _____ Е.В. Прохоров
подпись И.О. Фамилия

План выполнен полностью
(полностью, не полностью)

Руководитель работы « 06 » июня 2024 г. _____ А.С. Дорофеев
дата подпись И.О. Фамилия

Аннотация

Выпускная квалификационная посвящена разработке программного средства для проектирования модели данных. В ней подробно описан процесс разработки от изучения самого языка SQL и выбора стека технологий до прототипирования интерфейса, проектирования базы данных и конечно написания программного кода.

Каждый этап разработки выделен в отдельный пункт, который в свою очередь разделен на подпункты, что позволяет выработать понятную и чёткую структуру данной выпускной квалификационной работы.

Пояснительная записка разделена на такие части как:

- анализ языка SQL;
- выбор стека технология;
- проектирования интерфейса;
- создание программного средства;
- тестирование.

Данная работа имеет высокую актуальность так-как технология Structured Query Language используется в большинстве современных приложений, которые в той или ной форме работают с данными.

Были рассмотрены аналоги такие как:

- MySQL Workbench;
- Microsoft Visio;
- Lucidchart;
- Draw.io (diagrams.net);
- Dbdiagram.io.

Рассмотрение аналогов позволило создать уникальное приложение во многих аспектах, превосходящее свои аналоги.

Для разработки были выбраны React js для фронтенда и Node js для бэкенда одной из главных причин выбора такого стека была единая основа этих языков программирования java script, что упрощает переход между программирование фронтенда и бэкенда из-за высокой схожести их синтаксиса. Ещё стек из приведенных выше технологий является одним из самых актуальных и современных на данный момент. И отличается высокой степенью оптимизации, широкой степенью масштабируемости, что позволяет создать стабильно и быстро работающее веб-приложение.

Программные средства, которые использовались для разработки приложения Visual Studio Code для написания кода, Git для контроля версий программы и pgAdmin для работы с базой данных.

Также данная выпускная квалификационная работа содержит около 12 тысяч символов и 39 Рисунков, что позволяет говорить о высокой степени проработки и подробного описания разработки программного средства проектирования моделей данных.

Содержание

Введение

Актуальность проекта обусловлена тем, что в современном мире разработка приложений для работы с базами данных играет ключевую роль в обеспечении эффективного управления информацией. SQL (Structured Query Language) является одним из основных инструментов для работы с реляционными базами данных, позволяя разработчикам создавать, изменять и управлять данными. Однако, разработка SQL скриптов может быть сложной и трудоемкой задачей, особенно при работе с большими и сложными схемами баз данных.

Цель проекта состоит в том, чтобы создать инструмент, который позволит разработчикам легко проектировать базы данных, сокращая время и усилия, затрачиваемые на разработку SQL скриптов. Мы сосредотачиваемся на создании интуитивного пользовательского интерфейса, который позволит пользователям визуально создавать и модифицировать структуру баз данных, автоматически генерируя соответствующий SQL код.

Задачи, которые позволят достичь данной цели:

1. Анализ языка SQL
2. Выбор стека технологий
3. Проектирование интерфейса
4. Проектирование базы данных
5. Создание программного средства
6. Сравнение с аналогами
7. Тестирование

1 Анализ языка SQL

1.1 История языка SQL

История языка SQL (Structured Query Language) берет свое начало в 1970-х годах, когда IBM разрабатывал систему управления базами данных (СУБД) под названием System R. В это время проектировался язык запросов, который мог бы облегчить взаимодействие с данными в базе. Это привело к созданию языка SEQUEL (Structured English Query Language), который позднее был переименован в SQL из-за проблем с торговыми марками.

SQL был официально стандартизирован в 1986 году Американским Национальным Институтом Стандартов (ANSI) и в 1987 году Международной Организацией по Стандартизации (ISO). Эти стандарты были затем доработаны и дополнены в последующие годы.

С течением времени SQL стал широко распространенным языком запросов для реляционных баз данных. Различные вендоры СУБД, такие как Oracle, Microsoft, IBM, PostgreSQL и другие, развивали свои собственные реализации SQL, иногда с небольшими вариациями стандарта.

Важным моментом в истории SQL стало появление стандарта SQL-92, который включал множество новых функций и расширений языка. Это включало в себя поддержку хранимых процедур, транзакций, агрегатных функций и многих других возможностей.

В последующие годы SQL продолжал развиваться, появлялись новые стандарты, такие как SQL:1999, SQL:2003, SQL:2008 и т. д., каждый из которых добавлял новые функции и улучшения. Одним из важных направлений развития стало расширение возможностей SQL для работы с различными типами данных, включая XML, JSON и географические данные.

Сегодня SQL является одним из самых популярных и широко используемых языков запросов в мире. Он применяется в различных областях, включая веб-разработку, аналитику данных, бизнес-анализ, финансы, медицину и многое другое. SQL остается основным инструментом для работы с данными в реляционных базах данных и продолжает развиваться, чтобы удовлетворять растущие потребности современных информационных технологий.

SQL также стал основой для многих расширений и дополнительных инструментов, например, объектно-реляционные системы управления базами данных (ORDBMS), которые объединяют в себе возможности реляционных и объектно-ориентированных подходов к хранению данных. Такие системы, как PostgreSQL, предоставляют расширенные функциональные возможности, такие как пользовательские типы данных, методы и наследование, что делает их мощными инструментами для разработки сложных приложений.

Кроме того, эволюция SQL привела к появлению специализированных диалектов, таких как PL/SQL от Oracle и T-SQL от Microsoft SQL Server, которые предоставляют дополнительные функции и инструменты для разработки приложений и управления данными в пределах своих экосистем.

В современном мире SQL также стал интегрироваться с новыми технологиями, такими как машинное обучение и искусственный интеллект, что позволяет использовать его для анализа и обработки больших объемов данных, выявления паттернов и предсказания будущих событий.

Таким образом, SQL продолжает эволюционировать, адаптируясь к изменяющимся потребностям и технологическим трендам, и остается одним из ключевых инструментов в мире информационных технологий.

1.2 Oracle

История языка SQL в контексте Oracle включает в себя ряд ключевых моментов, начиная с момента создания самой компании.

Oracle Corporation была основана в 1977 году Ларри Эллисоном, Бобом Майнером и Эдом Остеном. Изначально компания занималась разработкой системы управления базами данных, которая впоследствии стала известной как Oracle Database.

Oracle была одним из первых вендоров, которые предложили поддержку языка SQL в своей СУБД. В начале 1980-х годов Oracle выпустила свою первую версию базы данных, которая включала поддержку SQL. Этот язык запросов стал ключевым элементом взаимодействия с данными в Oracle Database.

С развитием компании Oracle и ее продуктов SQL продолжал развиваться и совершенствоваться. Oracle активно участвовала в процессе стандартизации SQL, и многие возможности, которые стали стандартом в SQL, были первоначально внедрены именно в продуктах Oracle.

Одним из важных моментов в истории SQL Oracle было выпуск нескольких значительных версий своей базы данных, каждая из которых вносила новые функции и улучшения в язык SQL и его возможности. Например, Oracle Database 8i в конце 1990-х годов представила поддержку объектно-ориентированных возможностей в SQL, а Oracle Database 10g в начале 2000-х годов внедрила различные улучшения производительности и администрирования, включая автоматическое управление ресурсами и диагностику.

Следует отметить, что Oracle также разработала свои собственные расширения SQL, которые расширяют стандартные возможности языка для удовлетворения специфических потребностей пользователей и приложений.

Сегодня Oracle Database остается одной из самых популярных реляционных баз данных в мире, и SQL продолжает быть основным языком запросов для работы с данными в этой системе. Стандарт SQL, развиваемый международными организациями по стандартизации, и дальнейшие инновации Oracle в области SQL, продолжают формировать эволюцию языка запросов и его применение в современном мире информационных технологий.

Кроме того, в последние годы наблюдается рост популярности NoSQL баз данных, которые предлагают альтернативные модели хранения и обработки данных, отличные от традиционных реляционных подходов.

Несмотря на это, SQL остается востребованным и широко используемым, частично благодаря своей простоте и гибкости, а также широкому сообществу разработчиков и обширной документации.

Более того, SQL активно интегрируется с современными веб-технологиями, такими как фреймворки для разработки веб-приложений и системы управления контентом. Он играет важную роль в создании динамических веб-сайтов и приложений, обеспечивая эффективное хранение, извлечение и обработку данных.

В целом, SQL продолжает оставаться основой для управления данными в широком спектре областей, и его значимость и влияние продолжают расширяться с развитием информационных технологий.

1.3 Microsoft SQL Server (MS SQL)

История Microsoft SQL Server (MS SQL) связана с развитием компании Microsoft и их стремлением предоставить клиентам полноценное решение для управления базами данных под управлением их операционных систем.

Первая версия SQL Server была выпущена в 1989 году под названием SQL Server 1.0 для операционной системы OS/2, разработанной компанией IBM. Эта версия SQL Server была ориентирована на использование в клиент-серверных приложениях.

Следующая важная точка в истории MS SQL пришла в 1993 году с выпуском SQL Server 4.2, который был первой версией, выпущенной для платформы Windows NT. Это значительно расширило аудиторию продукта и укрепило его позиции на рынке.

За последующие годы SQL Server развивался, добавляя новые функции и улучшения. В 1995 году вышла версия SQL Server 6.0, в которой было внедрено множество новых возможностей, таких как хранимые процедуры, транзакции и триггеры.

С 2000 года Microsoft стала выпускать версии SQL Server с годовыми обновлениями. SQL Server 2000 был важным выпуском, который представил такие функции, как XML-поддержка, OLAP и аналитические возможности.

SQL Server 2005, выпущенный в 2005 году, был важным релизом, в котором были внедрены множество новых функций, включая улучшенную поддержку XML, встроенные отчеты и возможности для работы с большими объемами данных.

Дальнейшее развитие MS SQL продолжалось с выпуском SQL Server 2008, 2012, 2014, 2016 и последующих версий. Каждый релиз вносил новые функции и улучшения, такие как улучшенная производительность, расширенные возможности аналитики данных, поддержка облачных технологий и многое другое.

Одним из ключевых моментов в истории MS SQL был запуск SQL Server в облаке - SQL Azure, который предоставляет возможность использовать SQL Server в качестве облачной услуги.

Сегодня Microsoft SQL Server остается одним из ведущих реляционных СУБД на рынке. Он используется в широком спектре сфер, включая предприятия, веб-разработку, бизнес-аналитику и многое другое. SQL Server продолжает развиваться, внедряя новые технологии и функции, чтобы удовлетворить растущие потребности клиентов в области управления данными.

1.4 MySQL

История MySQL — это история создания одной из самых популярных и распространенных открытых реляционных баз данных в мире.

MySQL была разработана в 1994 году шведскими программистами Монти Виденусом и Давидом Акерсоном. Их целью было создание СУБД с открытым исходным кодом, которая была бы простой в использовании, масштабируемой и доступной для широкого круга пользователей.

Первая версия MySQL была выпущена в 1995 году. Она предлагала базовые функции реляционной базы данных, такие как хранение данных в таблицах, поддержку индексов и простые операции SELECT, INSERT, UPDATE и DELETE.

С течением времени MySQL быстро приобрела популярность благодаря своей простоте, производительности и надежности. Большой вклад в популяризацию MySQL внесла решительная поддержка открытого исходного кода и распространение под лицензией GPL (General Public License).

Ключевыми моментами в истории MySQL были выпуски новых версий с расширенным функционалом и улучшениями производительности. В 2000 году была выпущена версия MySQL 3.23, которая внесла значительные улучшения в производительность и функциональность запросов. MySQL 4.0, выпущенная в 2003 году, добавила поддержку хранимых процедур и триггеров, а также другие расширенные возможности.

Однако, возможно, наиболее значимым событием в истории MySQL было приобретение компанией Sun Microsystems в 2008 году. Это придавало MySQL дополнительные ресурсы и укрепляло ее позиции на рынке.

В 2010 году компания Oracle приобрела Sun Microsystems, включая MySQL. Это вызвало опасения в сообществе открытого исходного кода относительно будущего MySQL под управлением корпорации Oracle. В ответ на это была создана отдельная организация под названием MariaDB, форк MySQL, который продолжает развиваться независимо.

MySQL по-прежнему активно развивается, и ее последние версии включают в себя множество новых функций и улучшений, таких как поддержка JSON, улучшенные возможности репликации и масштабируемости, а также интеграция с облачными платформами.

Сегодня MySQL остается одной из самых популярных баз данных в мире и широко используется веб-разработчиками, компаниями и организациями различных масштабов. Ее открытый исходный код и богатый функционал

продолжают привлекать новых пользователей и обеспечивать ее популярность и востребованность в сообществе разработчиков.

1.5 PostgreSQL

История PostgreSQL (или просто Postgres) весьма интересна и связана с академическими исследованиями, разработкой исходного кода с открытым доступом и постоянным развитием сообщества.

Всё началось в 1986 году, когда проект Postgres был начат Майклом Стоунбрейкером, профессором информатики в Берклиском университете, и его командой студентов. Их целью было создание следующего поколения СУБД, которая была бы наследником управления данными в стиле Ingres (предшественника Postgres) и одновременно расширяла бы его функциональность.

Первая версия Postgres (Post Ingres) была выпущена в 1989 году. Это была объектно-реляционная база данных, предлагавшая множество инновационных функций, таких как поддержка пользовательских типов данных, подзапросов, правил и многое другое.

В 1996 году, после выпуска версии 6.0, проект был переименован в PostgreSQL, чтобы подчеркнуть его продолжающееся развитие и сближение с SQL-стандартами.

Одним из важных моментов в истории PostgreSQL было создание PostgreSQL Global Development Group в 1996 году, объединившей разработчиков и пользователей PostgreSQL со всего мира в единое сообщество. Это способствовало активному обмену знаниями и опытом, а также совместной разработке и совершенствованию СУБД.

В последующие годы PostgreSQL продолжала развиваться, добавляя новые функции и улучшения, такие как поддержка транзакций, сохраняемые процедуры, триггеры, репликация и многое другое. Важными моментами стали выпуск версий 7.0 в 2000 году, 8.0 в 2005 году и 9.0 в 2010 году, каждая из которых вносила значительные улучшения и инновации.

Сегодня PostgreSQL остается одной из самых мощных и функциональных открытых реляционных баз данных в мире. Ее использование распространено в различных областях, включая веб-разработку, аналитику данных, геоинформационные системы, финансовые приложения и многое другое. Сообщество PostgreSQL продолжает активно разрабатывать и совершенствовать СУБД, обеспечивая ее актуальность и конкурентоспособность в современном мире информационных технологий.

1.6 Синтаксис языка SQL

SQL, или язык структурированных запросов (Structured Query Language), это стандартный язык программирования для работы с реляционными базами данных. Он предоставляет набор инструкций для создания, изменения,

управления и извлечения данных из базы данных. Вот основные компоненты синтаксиса SQL:

1. Команды:
    - DDL (Data Definition Language): Команды для определения структуры базы данных, такие как CREATE, ALTER, DROP.
    - DML (Data Manipulation Language): Команды для управления данными, такие как SELECT, INSERT, UPDATE, DELETE.
    - DCL (Data Control Language): Команды для управления правами доступа к данным, такие как GRANT, REVOKE.
2. Ключевые слова:
    - SELECT: Извлечение данных из базы данных.
    - INSERT: Добавление данных в базу данных.
    - UPDATE: Обновление существующих данных в базе данных.
    - DELETE: Удаление данных из базы данных.
    - CREATE: Создание новых объектов базы данных, таких как таблицы, индексы и т. д.
    - ALTER: Изменение структуры объектов базы данных.
    - DROP: Удаление объектов базы данных.
    - И другие, в зависимости от конкретной реализации SQL.
3. Комментарии:
    - SQL поддерживает комментарии для документирования кода. Однострочные комментарии начинаются с двойного дефиса --, а многострочные комментарии заключаются между /* и */.
4. Таблицы:
    - База данных состоит из таблиц, которые хранят данные в виде строк и столбцов.
5. Столбцы:
    - Каждая таблица состоит из столбцов, которые определяют типы данных, которые могут храниться в каждой строке.
6. Строки:
    - Строки представляют отдельные записи или элементы данных в таблице.
7. Условия:
    - Условия используются для фильтрации данных в операторе SELECT с использованием ключевого слова WHERE.
8. Функции:
    - SQL предоставляет множество встроенных функций, таких как арифметические функции (SUM, AVG), функции работы со строками (CONCAT, UPPER), функции даты и времени (NOW, DATE_FORMAT) и т. д.
9. Соединения (JOIN):
    - SQL позволяет объединять данные из нескольких таблиц по определенным условиям с помощью оператора JOIN.

10.Группировка и агрегация:

- SQL поддерживает группировку данных с использованием оператора GROUP BY и агрегацию данных с использованием функций, таких как SUM, AVG, COUNT, MIN, MAX.

11.Сортировка:

- Для упорядочивания результатов запроса SQL используется оператор ORDER BY.

Это лишь краткий обзор основного синтаксиса SQL. Существует много различных реализаций SQL, таких как MySQL, PostgreSQL, SQLite, и каждая из них может иметь свои особенности и дополнения к стандарту SQL.

Пример оператора SQL вы можете увидеть на рисунке 1.

```
SELECT * FROM employees WHERE department = 'Sales' AND salary > 50000;
```

Рисунок 1 – Пример оператора

В этом примере:

- Оператор: SELECT * FROM employees WHERE department = 'Sales' AND salary > 50000;
- Предложение: SELECT * FROM employees WHERE department = 'Sales' AND salary > 50000;
- Таблица: employees
- Столбец: department, salary
- Константа: 'Sales', 50000

1.7 Диалекты SQL

Системы управления базами данных (СУБД) играют важную роль в современных информационных технологиях, предоставляя мощные инструменты для хранения, управления и обработки данных. SQL (Structured Query Language) является основным языком запросов, используемым для взаимодействия с данными в большинстве СУБД. Однако различные СУБД могут иметь свои собственные диалекты SQL, обладающие уникальными особенностями и нюансами.

Вот основные особенности диалектов SQL:

1. MySQL:
   - MySQL использует синтаксис с крайне гибкими кавычками, что позволяет использовать как обратные кавычки (`), так и обычные кавычки (').
   - Ограниченная поддержка оконных функций. Некоторые расширенные функции, такие как оконные функции, могут быть ограничены или недоступны.
   - Поддержка хранимых процедур и функций, но в сравнении с другими СУБД, такими как PostgreSQL, функциональность ограничена.
2. PostgreSQL:

- Полная поддержка стандарта SQL и расширенных функций. PostgreSQL предлагает богатый набор функций, включая оконные функции, общие таблицы выражений (CTE) и другие расширения.
- Большая гибкость в типах данных. PostgreSQL предоставляет широкий спектр типов данных, включая пользовательские типы и массивы.
- Поддержка транзакций и многопользовательской работы. PostgreSQL обеспечивает высокую степень надежности и целостности данных.

3. Microsoft SQL Server:
    - Использование ключевого слова **TOP** для ограничения числа возвращаемых строк.
    - Реализация оконных функций и аналитических функций начиная с версии SQL Server 2012.
    - Полная интеграция с другими продуктами Microsoft, такими как .NET Framework и Azure.
    - Поддержка транзакций и технологий репликации для обеспечения высокой доступности и масштабируемости.
4. Oracle:
    - Использование функции ROWNUM для ограничения числа возвращаемых строк, аналогично TOP в Microsoft SQL Server.
    - Мощная поддержка аналитических функций, включая PARTITION BY и ORDER BY.
    - Богатый набор типов данных, включая встроенные и пользовательские типы.
    - Поддержка пакетов и процедур PL/SQL (Procedural Language/Structured Query Language), что позволяет создавать хранимые процедуры и функции.
    - Высокая степень конфигурируемости и оптимизации запросов с использованием подсказок и индексов.
    - Обширные возможности для администрирования баз данных, включая управление пользователями, ролями, привилегиями и мониторингом производительности.

1.8 Скрипты для создания базы данных

Для эффективного создания базы данных необходимо разработать соответствующие скрипты, которые определят структуру хранения данных. Эти скрипты играют ключевую роль в процессе развертывания и поддержки приложений. Ниже приведены основные аспекты, которые следует учитывать при создании таких скриптов:

1. Создание структуры базы данных:

Скрипты для создания баз данных необходимы для определения структуры хранения данных в проекте. Это включает в себя определение таблиц, их столбцов, типов данных, ограничений, а также взаимосвязей между таблицами (например, внешние ключи).

2. Обеспечение целостности данных:

При создании таблиц в базе данных можно определить различные ограничения, такие как уникальность значений, ограничения на внешние ключи и т. д. Это позволяет обеспечить целостность данных, предотвращая некорректные или недопустимые значения.

3. Упрощение развертывания и миграций:

Создание скриптов для создания баз данных также упрощает процесс развертывания вашего проекта на новых серверах или его миграции на другие базы данных. Путем выполнения этих скриптов можно быстро восстановить структуру базы данных в новом окружении.

4. Документация и понимание проекта:

Скрипты для создания баз данных также могут служить документацией к вашему проекту, описывая его структуру и отношения между данными. Это упрощает понимание проекта другими разработчиками или членами вашей команды.

2 Сравнение с аналогами

1. MySQL Workbench:

Плюсы:

- Мощный функционал для проектирования баз данных, включая создание схем, определение отношений, генерацию SQL-кода и т. д.
- Интеграция с MySQL, что облегчает развертывание и управление базами данных.
- Бесплатность и открытый исходный код.

Минусы:

- Ограниченность использования только с MySQL, что может быть недостаточно для проектов, использующих другие СУБД.
- Интерфейс может показаться сложным для новичков из-за большого количества функций.

2. Microsoft Visio:

Плюсы:

- Широкий спектр возможностей для создания различных типов диаграмм, включая диаграммы баз данных.
- Интеграция с другими продуктами Microsoft, такими как Microsoft Office.

Минусы:

- Платная лицензия, что может быть проблемой для некоммерческих или индивидуальных пользователей.
- Не является специализированным инструментом для проектирования баз данных, что может привести к ограничениям в функциональности.

3. Lucidchart:

Плюсы:

- Онлайн-приложение с удобным интерфейсом, доступным из любого браузера.
- Различные шаблоны и элементы для создания ER-диаграмм и других типов диаграмм.

Минусы:

- Ограничения в бесплатной версии, включая ограниченное количество диаграмм и отсутствие некоторых функций.
- Не такой широкий набор функций для проектирования баз данных, как у специализированных инструментов.

4. draw.io (diagrams.net):

Плюсы:

- Бесплатное и открытое программное обеспечение.
- Широкий набор элементов для создания диаграмм, включая ER-диаграммы.
- Возможность экспорта в различные форматы.

Минусы:

- Отсутствие интеграции с конкретными СУБД, что требует ручного создания SQL-кода.
- Отсутствие онлайн-синхронизации и совместной работы в реальном времени в бесплатной версии.

5. Dbdiagram.io:

Плюсы:

- Простота использования благодаря использованию языка разметки Markdown.
- Онлайн-инструмент с возможностью доступа из любого браузера.

Минусы:

- Ограниченные возможности по сравнению с более продвинутыми инструментами, такими как MySQL Workbench или Lucidchart.
- Отсутствие некоторых функций, таких как генерация SQL-кода, в бесплатной версии.

Преимуществом нашего приложения над вышеуказанными является.

1. Бесплатность
2. Открытый исходный код
3. Возможность генерировать SQL код
4. Поддержка различных языков SQL
5. Интуитивно понятный интерфейс

3 Выбор стека технологий

Наш проект представляет собой веб-приложение, созданное с целью облегчить процесс создания SQL скриптов. Предполагается, что данное приложение будет использоваться как для разработчиков, так и для аналитиков данных, упрощая им работу с базами данных. При разработке данного приложения мы учитываем ряд ключевых требований, которые должны быть удовлетворены для обеспечения эффективной работы и удовлетворения потребностей пользователей:

- Производительность: Приложение должно обеспечивать высокую производительность и отзывчивость интерфейса даже при работе с большими объемами данных.
- Масштабируемость: Необходимо, чтобы приложение могло масштабироваться в случае увеличения нагрузки или расширения функционала.
- Безопасность: Важно обеспечить защиту данных пользователей и приложения от угроз безопасности.
- Интуитивный интерфейс: Приложение должно иметь понятный и удобный интерфейс для пользователей всех уровней навыков.
- Интеграция: Возможность интеграции с другими инструментами разработки, такими как системы управления версиями и проектными досками.

3.1 Backend

Backend, или серверная часть, представляет собой основную часть веб-приложения, ответственную за обработку запросов от клиентской части (frontend) и взаимодействие с базами данных, внешними сервисами и другими компонентами системы. Он обеспечивает логику приложения, обработку данных, аутентификацию и авторизацию пользователей, а также другие бизнес-логику, необходимую для функционирования приложения.

Выбор технологий для backend-разработки играет ключевую роль в определении производительности, масштабируемости, безопасности и функциональности веб-приложения. При рассмотрении различных вариантов для backend, важно учитывать требования проекта и потребности пользователей, чтобы выбрать наиболее подходящий технологический стек.

Выбор Node.js в качестве бэкэнд-технологии для нашего проекта можно обосновать следующим образом:

1. Высокая производительность: Node.js построен на основе событийно-ориентированной архитектуры и асинхронного ввода-вывода, что делает его очень эффективным в обработке большого количества одновременных запросов. Это особенно полезно в приложениях с высокой нагрузкой.

2. Единый язык программирования: Использование JavaScript как языка программирования как на клиентской, так и на серверной стороне позволяет уменьшить затраты на обучение и разработку, а также обеспечивает единый стиль кода и переиспользование некоторых компонентов.
3. Большое сообщество и экосистема: Node.js имеет огромное сообщество разработчиков и обширную экосистему библиотек и фреймворков, которые упрощают разработку и расширение функциональности ваших приложений.
4. Модульность и гибкость: Node.js поощряет модульную архитектуру приложений, что позволяет разрабатывать приложения из множества маленьких и переиспользуемых компонентов. Это делает код более чистым, поддерживаемым и масштабируемым.
5. Быстрый старт: Node.js предлагает легкий и быстрый способ создания прототипов приложений благодаря своей простоте и минималистичности.

Сравнение с аналогами:

1. Java (Spring Boot):
   - Java предлагает высокую производительность и надежность, особенно для крупных корпоративных приложений.
   - Spring Boot, в частности, предоставляет множество инструментов и функций для быстрого создания и развертывания приложений.
   - Однако Java имеет более высокий порог входа из-за необходимости в компиляции и более объемного кода.
2. Python (Django или Flask):
   - Python также является популярным выбором для бэкэнд-разработки благодаря своей простоте и выразительности.
   - Django и Flask предоставляют мощные инструменты и фреймворки для создания веб-приложений на Python.
   - Однако Python может быть менее эффективным в обработке большого количества одновременных запросов из-за своего многопоточного подхода.
3. Ruby (Ruby on Rails):
   - Ruby on Rails предоставляет быстрый способ создания веб-приложений с помощью принципа "соглашение больше, чем конфигурация" и обширной библиотеки готовых решений.
   - Ruby может быть привлекательным выбором для команд, предпочитающих элегантный и выразительный код.
   - Однако Ruby может быть менее эффективным в обработке большого количества одновременных запросов из-за своей медленной скорости выполнения.

В целом, выбор между Node.js и его аналогами зависит от конкретных требований вашего проекта, предпочтений команды разработчиков и контекста приложения. Node.js отлично подходит для создания быстрых и

масштабируемых веб-приложений с высокой производительностью и удобством разработки.

3.2 Frontend

Frontend, или клиентская часть, представляет собой интерфейс веб-приложения, с которым взаимодействуют пользователи. Он отвечает за отображение данных, интерактивность и визуальное взаимодействие пользователя с приложением.

Выбор технологий для frontend-разработки имеет решающее значение для создания удобного, функционального и привлекательного пользовательского интерфейса. При выборе технологического стека для frontend необходимо учитывать требования проекта к дизайну, производительности, поддержке различных устройств и браузеров, а также опыт пользователя.

Ключевые факторы, которые следует учитывать при выборе технологий для frontend-разработки, включают в себя производительность, масштабируемость, поддержку современных стандартов веб-разработки, а также удобство использования и обучения для разработчиков.

Выбор React.js в качестве фронтенд-технологии для нашего проекта также может быть обоснован несколькими факторами:

1. Производительность и эффективность: React.js использует виртуальный DOM и механизм перерисовки только измененных компонентов, что обеспечивает высокую производительность и эффективное использование ресурсов браузера.
2. Компонентный подход: React.js основан на компонентах, что позволяет разрабатывать приложения из небольших и переиспользуемых элементов. Это делает код более организованным, легко поддерживаемым и масштабируемым.
3. Односторонний поток данных: React.js пропагандирует однонаправленный поток данных (от родительских компонентов к дочерним), что облегчает понимание и отслеживание данных в приложении и упрощает управление состоянием.
4. Широкая экосистема: React.js имеет обширную экосистему инструментов, библиотек и фреймворков, таких как Redux, React Router, Material-UI и многие другие, которые упрощают разработку и расширение функциональности приложений.
5. JSX синтаксис: React.js использует JSX - расширение JavaScript, позволяющее писать HTML-подобный код внутри JavaScript. Это делает код более декларативным, понятным и удобным для работы.
6. Виртуализация на стороне клиента: React.js позволяет создавать мощные интерактивные интерфейсы, включая сложные веб-приложения с асинхронной загрузкой данных и динамическим обновлением пользовательского интерфейса.

Сравнение с аналогами:

1. Angular:
    - Angular также является популярным фронтенд-фреймворком, предоставляющим множество инструментов и функций для создания веб-приложений.
    - Однако Angular имеет более высокий порог входа из-за своей сложной архитектуры и использования TypeScript.
2. Vue.js:
    - Vue.js - это еще один современный фронтенд-фреймворк, который обеспечивает легкую изучаемость и простоту в использовании, а также поддержку компонентного подхода.
    - Однако React.js часто предпочтительнее для крупных и сложных проектов благодаря своей более широкой экосистеме и поддержке со стороны крупных компаний.
3. Svelte:
    - Svelte предлагает новый подход к созданию веб-приложений, основанный на компиляции компонентов в чистый JavaScript во время сборки.
    - Хотя Svelte обещает лучшую производительность и меньший объем кода, React.js все еще остается более распространенным и широко используемым фреймворком.

В целом, React.js представляет собой мощный инструмент для создания современных веб-приложений, обладающий высокой производительностью, гибкостью и широкой поддержкой сообщества.

3.3 База данных

База данных представляет собой структурированное хранилище данных, которое используется для хранения, управления и организации информации, необходимой для функционирования приложений и систем. Выбор технологий для баз данных играет важную роль в обеспечении эффективного хранения, доступа и обработки данных.

При выборе базы данных необходимо учитывать требования проекта к масштабируемости, производительности, надежности, безопасности и поддержке специфических типов данных. Важно также оценить потенциальные интеграционные возможности с другими компонентами системы и удобство использования для разработчиков.

Ключевые факторы, которые следует учитывать при выборе технологии базы данных, включают в себя тип данных, модель хранения, поддержку транзакций, масштабируемость, производительность, а также возможности резервного копирования и восстановления данных.

Выбор PostgreSQL в качестве основной базы данных для нашего проекта можно обосновать по ряду причин:

1. Многоплатформенность: VS Code поддерживает операционные системы Windows, macOS и Linux, что делает его универсальным инструментом для разработчиков, работающих на различных платформах.

2. Расширяемость: VS Code предлагает обширный репозиторий расширений, который позволяет разработчикам настраивать среду разработки под свои потребности. От поддержки различных языков программирования до инструментов управления версиями и отладки, расширения обеспечивают широкий спектр функциональности.
3. Интеграция с Git: VS Code имеет встроенную поддержку Git, что делает управление версиями и совместную работу в проектах легкой и интуитивно понятной.
4. Мощный редактор кода: Редактор кода в VS Code обладает множеством возможностей, включая подсветку синтаксиса, автоматическое завершение кода, быструю навигацию, интегрированный поиск и замену текста, а также поддержку различных видов файлов.
5. Отладка: Интегрированная система отладки в VS Code облегчает процесс обнаружения и исправления ошибок в коде.
6. Интеграция с различными фреймворками и средами разработки: VS Code поддерживает множество популярных языков программирования, фреймворков и инструментов разработки, таких как JavaScript, Python, Node.js, .NET и многие другие.
7. Автоматические обновления: VS Code регулярно обновляется, предоставляя пользователям новые функции и улучшения без необходимости установки новой версии.

Учитывая эти факторы, PostgreSQL представляет собой привлекательное решение для многих проектов, особенно тех, где требуется надежная, масштабируемая и гибкая база данных.

Давайте сравним PostgreSQL с двумя из его основных аналогов - MySQL и SQLite - по нескольким ключевым аспектам:

1. Функциональные возможности:
    - PostgreSQL: Обладает богатым набором функциональных возможностей, включая поддержку геоданных, полнотекстовый поиск, триггеры, процедуры, расширяемые типы данных и многое другое.
    - MySQL: Предлагает широкий набор стандартных функций и возможностей, но несколько более ограничен в расширяемости и функциональности по сравнению с PostgreSQL.
    - SQLite: Легкая встраиваемая база данных, обычно используется для простых приложений или встраивается в мобильные приложения. Несмотря на это, она поддерживает большинство стандартных SQL-возможностей.
2. Производительность и масштабируемость:
    - PostgreSQL: Обеспечивает хорошую производительность и масштабируемость, особенно при правильной настройке индексов и оптимизации запросов. Может быть использован для крупных и сложных проектов.

- MySQL: Имеет хорошую производительность и масштабируемость, особенно на низкой и средней нагрузке. Однако при очень высоких нагрузках может потребоваться более тщательная настройка.
- SQLite: Часто используется для небольших приложений или для прототипирования из-за своей простоты и легковесности. Не подходит для высоконагруженных приложений или крупных баз данных.

3. Поддержка стандартов и соответствие SQL:
   - PostgreSQL: Стремится к полному соответствию стандартам SQL и предоставляет обширные возможности для разработчиков.
   - MySQL: Хорошо соответствует основным стандартам SQL, но может отличаться в некоторых аспектах от PostgreSQL.
   - SQLite: Также соответствует основным стандартам SQL, но, как и в случае с MySQL, есть некоторые различия в функциональности и возможностях.
4. Распространенность и экосистема:
   - PostgreSQL: Стремится к полному соответствию стандартам SQL и предоставляет обширные возможности для разработчиков.
   - MySQL: Хорошо соответствует основным стандартам SQL, но может отличаться в некоторых аспектах от PostgreSQL.
   - SQLite: Также соответствует основным стандартам SQL, но, как и в случае с MySQL, есть некоторые различия в функциональности и возможностях.

В целом, PostgreSQL представляет собой мощную и гибкую базу данных, особенно подходящую для крупных и сложных проектов, требующих богатый набор функциональных возможностей и высокую надежность. MySQL и SQLite также имеют свои преимущества и подходят для различных типов приложений и сценариев использования.

Безопасность баз данных — это критически важный аспект в области информационной технологии. Она охватывает различные аспекты, включая аутентификацию, авторизацию, шифрование, аудит и другие меры для защиты данных от несанкционированного доступа, изменений и утечек.

1. Аутентификация

Аутентификация — это процесс проверки подлинности пользователей и устройств перед предоставлением доступа к базе данных. Включает в себя методы, такие как:

- Имя пользователя и пароль: Самый распространенный метод аутентификации, который требует от пользователей предоставить учетные данные для доступа к базе данных.
- Многофакторная аутентификация (MFA): Дополнительный слой защиты, который требует от пользователя предоставить не только

пароль, но и другой аутентификационный фактор, такой как одноразовый код, биометрические данные или аппаратный ключ.
- Интеграция с внешними системами аутентификации: Возможность использовать сторонние системы аутентификации, такие как LDAP, OAuth или SAML, для централизованного управления доступом.

2. Авторизация

Авторизация определяет права доступа пользователей к данным и операциям в базе данных. Ключевые концепции включают:
- Ролевая модель доступа: Определение ролей и привилегий для пользователей и групп пользователей. Назначение ролей облегчает управление доступом и обеспечивает принцип наименьших привилегий (Principle of Least Privilege).
- Гибкие политики доступа: Возможность определения детализированных прав доступа на уровне таблиц, столбцов, процедур и других объектов базы данных.

3. Шифрование данных

Шифрование данных — это процесс преобразования данных в нечитаемый формат с использованием криптографических алгоритмов. Основные виды шифрования включают:
- Шифрование в покое: Защита данных в хранилище базы данных путем шифрования файлов данных и журналов транзакций.
- Шифрование в движении: Обеспечение безопасной передачи данных между клиентами и серверами с использованием протоколов шифрования, таких как SSL / TLS.

4. Аудит и мониторинг

Аудит и мониторинг — это процесс записи и анализа действий пользователей и изменений данных в базе данных. Это позволяет выявлять несанкционированные действия и угрозы безопасности. Основные аспекты включают:
- Журналирование событий: Запись всех действий пользователей, выполненных запросов и изменений данных для последующего анализа и отслеживания.
- Анализ журналов: Использование специализированных инструментов для обнаружения аномалий, внутренних угроз и других подозрительных действий.

3.4 Инструменты разработки

Выбор правильных инструментов разработки играет ключевую роль в эффективной разработке программного обеспечения, обеспечивая комфортную рабочую среду для разработчиков и оптимизируя процессы создания, тестирования и развертывания приложений.

При рассмотрении инструментов разработки необходимо учитывать требования проекта, особенности команды разработки и конкретные потребности разработчиков. Это включает в себя выбор интегрированных сред разработки (IDE), систем управления версиями кода, инструментов для непрерывной интеграции и развертывания (CI/CD), а также других инструментов для управления проектом и коммуникации в команде.

Целью данного раздела является обсуждение ключевых инструментов разработки, которые могут быть использованы для создания нашего проекта, а также их преимуществ и сферы применения.

Visual Studio Code (VS Code) — это мощное, легкое и гибкое интегрированное средство разработки (IDE), созданное Microsoft. Вот несколько ключевых особенностей:

1. Многоплатформенность: VS Code поддерживает операционные системы Windows, macOS и Linux, что делает его универсальным инструментом для разработчиков, работающих на различных платформах.
2. Расширяемость: VS Code предлагает обширный репозиторий расширений, который позволяет разработчикам настраивать среду разработки под свои потребности. От поддержки различных языков программирования до инструментов управления версиями и отладки, расширения обеспечивают широкий спектр функциональности.
3. Интеграция с Git: VS Code имеет встроенную поддержку Git, что делает управление версиями и совместную работу в проектах легкой и интуитивно понятной.
4. Мощный редактор кода: Редактор кода в VS Code обладает множеством возможностей, включая подсветку синтаксиса, автоматическое завершение кода, быструю навигацию, интегрированный поиск и замену текста, а также поддержку различных видов файлов.
5. Отладка: Интегрированная система отладки в VS Code облегчает процесс обнаружения и исправления ошибок в коде.
6. Интеграция с различными фреймворками и средами разработки: VS Code поддерживает множество популярных языков программирования, фреймворков и инструментов разработки, таких как JavaScript, Python, Node.js, .NET и многие другие.
7. Автоматические обновления: VS Code регулярно обновляется, предоставляя пользователям новые функции и улучшения без необходимости установки новой версии.

Эти особенности делают Visual Studio Code одним из самых популярных и мощных инструментов разработки для широкого круга разработчиков.

Git — это распределенная система управления версиями, разработанная Линусом Торвальдсом. Вот некоторые ключевые характеристики и преимущества Git:

1. Многоплатформенность: VS Code поддерживает операционные системы Windows, macOS и Linux, что делает его универсальным инструментом для разработчиков, работающих на различных платформах.
2. Расширяемость: VS Code предлагает обширный репозиторий расширений, который позволяет разработчикам настраивать среду разработки под свои потребности. От поддержки различных языков программирования до инструментов управления версиями и отладки, расширения обеспечивают широкий спектр функциональности.
3. Интеграция с Git: VS Code имеет встроенную поддержку Git, что делает управление версиями и совместную работу в проектах легкой и интуитивно понятной.
4. Мощный редактор кода: Редактор кода в VS Code обладает множеством возможностей, включая подсветку синтаксиса, автоматическое завершение кода, быструю навигацию, интегрированный поиск и замену текста, а также поддержку различных видов файлов.
5. Отладка: Интегрированная система отладки в VS Code облегчает процесс обнаружения и исправления ошибок в коде.
6. Интеграция с различными фреймворками и средами разработки: VS Code поддерживает множество популярных языков программирования, фреймворков и инструментов разработки, таких как JavaScript, Python, Node.js, .NET и многие другие.
7. Автоматические обновления: VS Code регулярно обновляется, предоставляя пользователям новые функции и улучшения без необходимости установки новой версии.

Git стал стандартом для управления версиями в различных проектах благодаря своей гибкости, мощным возможностям и широкому распространению.

pgAdmin - это бесплатное кроссплатформенное программное обеспечение с открытым исходным кодом, предназначенное для администрирования PostgreSQL и связанных баз данных. Оно обеспечивает удобный графический интерфейс для управления базами данных PostgreSQL и выполнения различных административных задач.

Вот некоторые основные черты и возможности pgAdmin:

1. Интерфейс с поддержкой многих окон: pgAdmin предоставляет удобный пользовательский интерфейс, который позволяет администраторам работать с несколькими окнами и вкладками, что облегчает управление несколькими базами данных одновременно.
2. Обозреватель объектов: В pgAdmin есть обозреватель объектов, который позволяет администраторам легко просматривать, создавать, изменять и удалять различные объекты базы данных, такие как таблицы, представления, индексы, функции и т.д.
3. Редактор SQL-запросов: pgAdmin включает мощный SQL-редактор с подсветкой синтаксиса, автозавершением и другими полезными

функциями, что делает написание и выполнение SQL-запросов более удобным.

4. Административные инструменты: pgAdmin предоставляет различные административные инструменты для управления базой данных, такие как мониторинг активности, администрирование безопасности, настройка параметров и т.д.
5. Интеграция с сервером: pgAdmin обеспечивает интеграцию с сервером PostgreSQL, что позволяет администраторам легко подключаться к удаленным серверам и управлять ими из графического интерфейса.
6. Поддержка расширений и плагинов: pgAdmin расширяем и поддерживает плагины, что позволяет пользователю добавлять новые функции и возможности в программу.

В целом, pgAdmin является мощным и гибким инструментом для администрирования баз данных PostgreSQL, который обеспечивает широкий спектр функций и возможностей для работы с данными и объектами базы данных.

4 Проектирование базы данных

Web-приложению необходима база данных для хранения записей авторизованных пользователей и информации, необходимой для функционирования web-приложения, такой как названия SQL, доступные на нашем ресурсе.

База данных будет играть ключевую роль в обеспечении безопасности и управлении доступом к нашим ресурсам. Она будет содержать информацию о пользователях, их учетных данных, а также о том, какие SQL-запросы доступны на web-приложении.

Эта база данных будет служить основой для аутентификации пользователей, контроля доступа и обеспечения целостности данных на web-приложении. Благодаря ей, мы сможем эффективно управлять информацией, предоставлять пользователям необходимые функции и обеспечивать безопасность данных.

Разработка и поддержка такой базы данных требует внимательного проектирования, чтобы обеспечить оптимальную производительность, безопасность и масштабируемость нашего веб-ресурса.

Определим сущности и атрибуты базы данных.

Сущность представляет собой объект или концепцию в вашей системе, который вы хотите отслеживать и хранить в базе данных. Это может быть что угодно, от конкретного объекта (например, человек, товар, заказ) до абстрактной концепции (например, категория товара, роль пользователя). Сущности обычно представлены в виде таблиц в базе данных.

Атрибуты — это свойства или характеристики сущности, которые определяют её. Например, если рассматривается сущность "пользователь", то её атрибутами могут быть имя, фамилия, адрес электронной почты и т.д. Атрибуты помогают описать каждую сущность более детально и хранятся в виде столбцов в таблицах базы данных их можно увидеть в таблице 1.

Таблица 1 – Сущности и атрибуты

Сущность	Атрибты	
Пользоватиели	Логин	Пароль
Список SQL	Название	
Таблица	Поле	Название

Далее нормализуем данные.

Нормализация данных — это процесс организации структуры базы данных таким образом, чтобы минимизировать избыточность информации и предотвратить аномалии при вставке, обновлении и удалении данных. Цель нормализации состоит в том, чтобы разделить данные на небольшие логически связанные таблицы, уменьшив повторение информации и обеспечив согласованность данных.

Основные преимущества нормализации данных включают уменьшение избыточности информации, облегчение поддержки и модификации базы данных, а также предотвращение аномалий при манипуляции данными.

Однако следует помнить, что слишком агрессивная нормализация может привести к усложнению запросов и ухудшению производительности, поэтому важно достигать баланса между нормализацией и денормализацией в зависимости от конкретных потребностей приложения.

Чтобы обеспечить нормальное функционирование и редактирование данных из сущности таблица было выделенных целых три таблицы, которые связаны между собой внешними ключами. Смотрите на рисунке 2

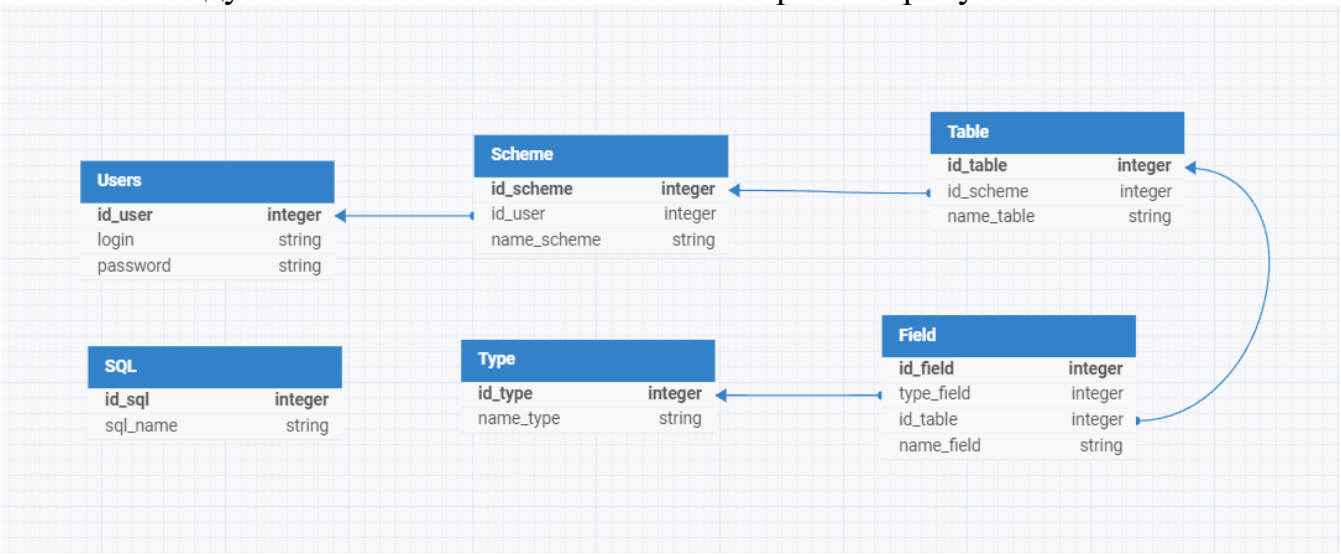


Рисунок 2 – Схема данных

Начнём с разбора первой таблицы Users, которую можно увидеть рисунок 3, в ней находятся данные, которые понадобятся для авторизации соответственно логин и пароль.

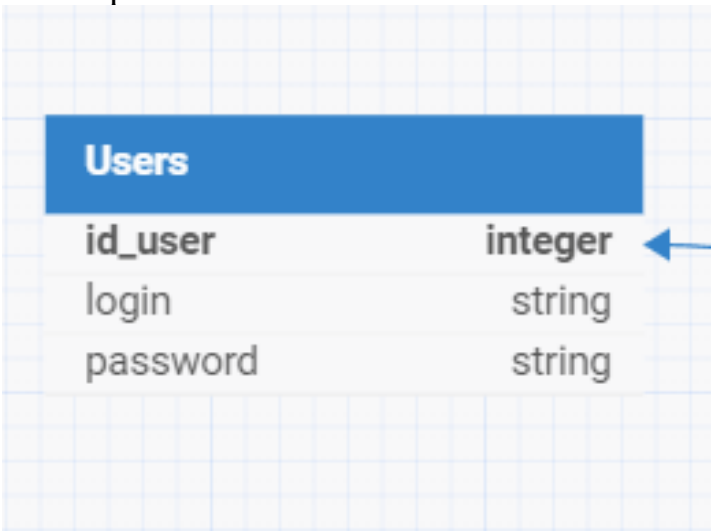


Рисунок 3 – Таблица Users

В следующей табличке хранится название схемы, и внешний ключ id пользователя, который позволит узнать какой пользователь создал данную схему. Её можно увидеть на рисунке 4.

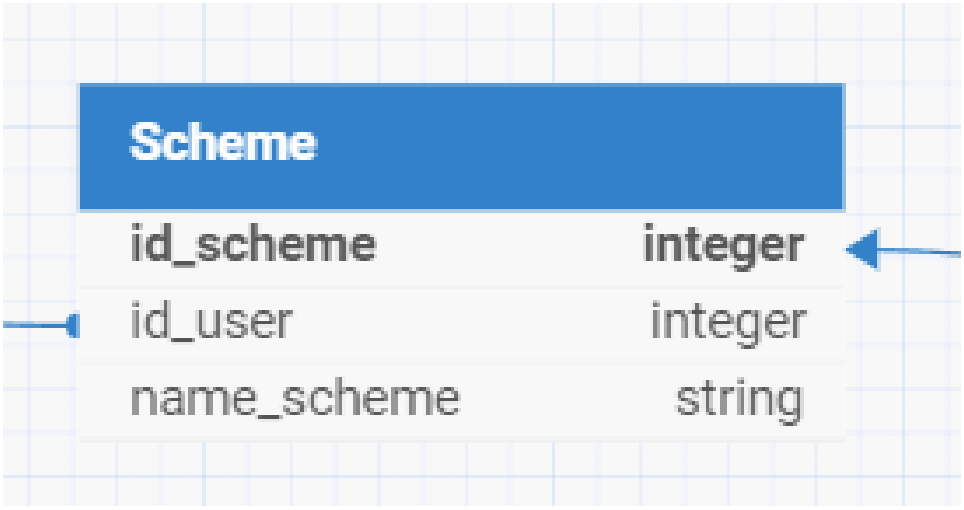


Рисунок 4 – Таблица Scheme

Далее идёт таблица, которую можно увидеть на рисунке 5, где хранятся данные о таблицах, которые хранятся в этой схеме. Это осуществляется с помощь. внешнего ключа id схемы.

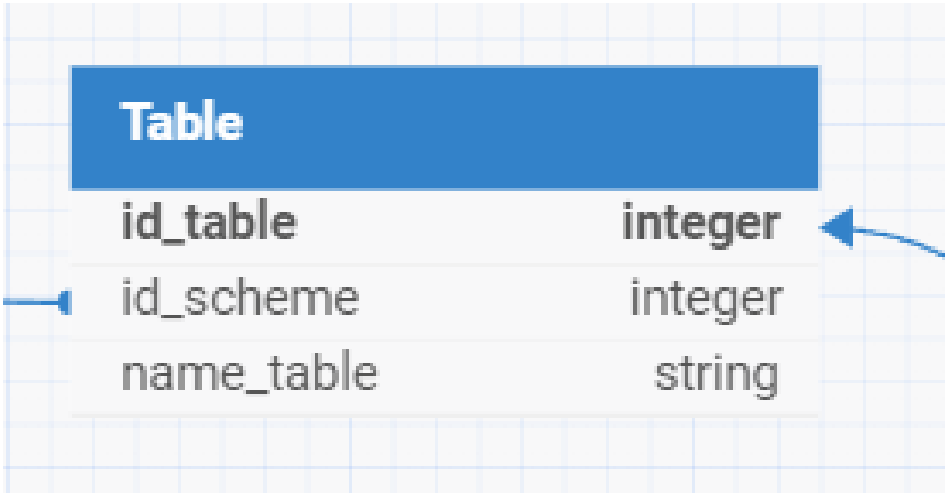


Рисунок 5 – Таблица Table

Ещё правее находится таблица со строками нашей таблицы, там хранится тип строки и её имя, а еще внешний ключ, который связывает её с таблицей в которой находятся эти строки. Её можно увидеть на рисунке 6.

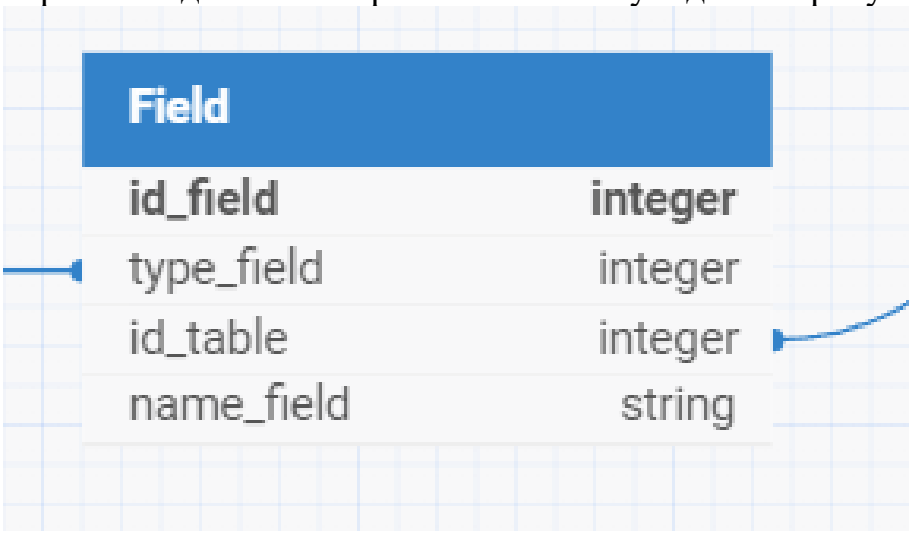


Рисунок 6 – Таблица Field

Далее идёт табличка, в которой хранятся типы данных наших строк, которую можно увидеть на рисунке 7.

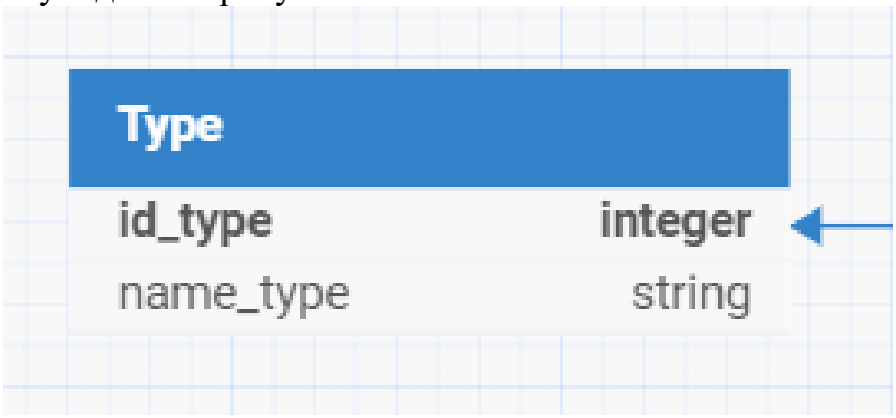


Рисунок 7 – Таблица Type

Отдельной не связанной с другими таблицами внешними ключами является таблица с типами SQL, которые поддерживает наше web-приложение. Её можно увидеть на рисунке 8.

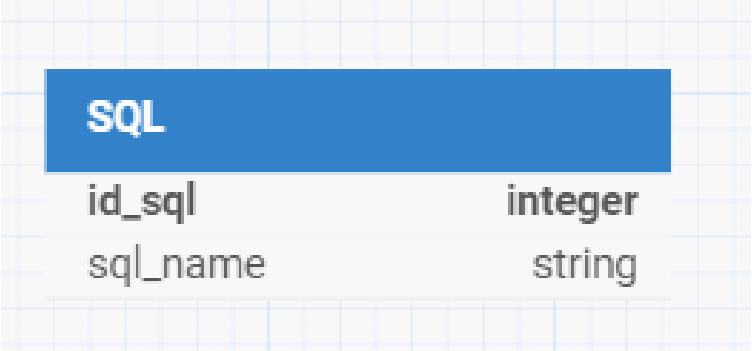


Рисунок 8 – Таблица SQL

Такая структура данных позволит эффективно сохранять записи пользователей и хранить их длительный срок.

В таблице 2 представлен перевод и название полей таблиц нашей базы данных.

Таблица 2 – Назначение таблиц

Таблица	Поле	Назначение
Users (Пользователи)	id_user (ид пользователя)	Уникальный идентификатор каждого пользователя
	login(логин)	Уникальное имя пользователя
	password(пароль)	Пароль от аккаунта
Scheme (Схема)	id_scheme (ид схемы)	Уникальный идентификатор каждой схемы
	id_user (ид пользователя)	Уникальный идентификатор каждого пользователя
	name_scheme (название схемы)	Название схемы
Table (Таблица)	id_table (ид таблицы)	Уникальный идентификатор каждой таблицы
	id_scheme (ид схемы)	Уникальный идентификатор каждой схемы
	name_table (название таблицы)	Название таблицы
Field (Поле)	id_field	Уникальный идентификатор каждого поля
	type_field (тип поля)	Тип поля
	id_table (ид таблицы)	Уникальный идентификатор каждой таблицы
	name_field (название поля)	Название поля
Type (тип)	id_type (ид типа)	Уникальный идентификатор каждого типа
	name_type (название типа)	Название типа
SQL (эскьюэль)	id_sql (ид sql)	Уникальный идентификатор каждого SQL
	sql_name (название sql)	Название SQL

5 Проектирования интерфейса

Проектирование интерфейса играет фундаментальную роль в разработке веб-приложений, определяя пользовательский опыт и взаимодействие с приложением. Этот раздел посвящен обсуждению процесса проектирования интерфейса, включая исследование и анализ, прототипирование. Рассмотрение данных аспектов поможет создать удобный, интуитивно понятный и привлекательный интерфейс, способствующий успешной реализации задач приложения и удовлетворению потребностей пользователей.

5.1 Исследование и анализ

Идентификация целевой аудитории:

1. Студенты и обучающиеся:
    - Эта категория пользователей включает студентов университетов и колледжей, изучающих базы данных, а также людей, проходящих курсы и обучение по данной тематике.
    - Они ищут инструменты, которые помогут им понять основы проектирования баз данных и практиковаться в создании собственных схем.
2. Разработчики программного обеспечения:
    - Разработчики, создающие приложения и сервисы, которые используют базы данных в качестве хранилища данных.
    - Они нуждаются в инструментах для проектирования и моделирования структуры баз данных перед началом разработки.
3. Баз данных администраторы:
    - Специалисты, отвечающие за управление и поддержку баз данных в предприятии.
    - Им требуются инструменты для анализа существующих баз данных, создания новых схем и оптимизации их производительности.
4. Аналитики данных:
    - Специалисты, занимающиеся анализом данных и созданием отчетов на основе информации из баз данных.
    - Они могут использовать инструменты для создания моделей данных и определения требований к структуре баз данных для оптимального анализа.
5. Начинающие пользователи:
    - Люди, не имеющие опыта в проектировании баз данных, но имеющие потребность в создании простых структур для своих проектов или идей.
    - Им нужен интуитивно понятный и легко осваиваемый инструмент для создания баз данных без необходимости в глубоких знаниях теории баз данных.

Анализ требований для приложения проектирования баз данных:

1. Создание и редактирование таблиц:
    - Пользователи должны иметь возможность создавать новые таблицы для хранения данных.
    - Требуется возможность добавления и удаления столбцов в таблицах, а также изменения их типов данных и других свойств.
2. Определение отношений между таблицами:
    - Приложение должно позволять пользователям определять связи между различными таблицами, такие как один-к-одному, один-ко-многим, многие-ко-многим.
    - Должна быть возможность управления связями, включая их создание, изменение и удаление.
3. Генерация SQL-кода:
    - Пользователи должны иметь возможность генерировать SQL-код для создания базы данных на основе созданных ими таблиц и связей.
    - Код должен быть совместим с распространенными СУБД, такими как MySQL, PostgreSQL, SQLite и другими.
4. Визуализация структуры базы данных:
    - Приложение должно предоставлять графический интерфейс для визуализации структуры базы данных, включая таблицы и связи между ними.
    - Требуется возможность масштабирования и перемещения элементов для удобного просмотра.
5. Поддержка различных типов данных:
    - Приложение должно поддерживать широкий спектр типов данных, используемых в различных СУБД, включая числовые, текстовые, даты, времена и другие.
    - Должна быть возможность настройки дополнительных параметров для каждого типа данных, таких как размер поля, ограничения на значения и т. д.
6. Импорт и экспорт данных:
    - Пользователи должны иметь возможность импортировать данные из внешних источников, таких как CSV-файлы или другие базы данных.
    - Требуется возможность экспорта структуры базы данных и ее содержимого для обмена данными с другими приложениями или для создания резервных копий.
7. Удобный интерфейс пользователя:
    - Интерфейс приложения должен быть интуитивно понятным и легко освоимым даже для пользователей без опыта в проектировании баз данных.
    - Требуется обеспечить удобство взаимодействия с элементами интерфейса, такими как контекстные меню, кнопки быстрого доступа и т. д.

8. Поддержка коллаборации:
    - Приложение должно предоставлять возможность совместной работы нескольких пользователей над одной базой данных, в том числе с возможностью совместного редактирования и комментирования структуры.
    - Требуется механизм управления доступом и версионирования изменений.

5.2 Прототипирование

Начнём с базового прототипа нашего интерфейса. Прототип интерфейса можно увидеть на рисунке 9.

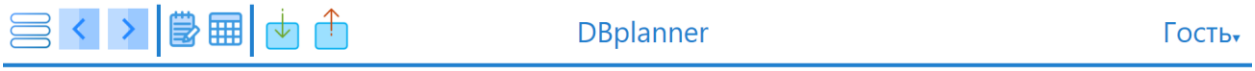


Рисунок 9 – Прототип интерфейса

Далее разберем его отдельные элементы.

В низу находится Footer с кнопками изменения масштаба, который можно увидеть на рисунке 10.

Рисунок 10 – Прототип Footer

Рисунок 11 – Прототип кнопок масштабирования

Кнопки масштабирования нужны, чтобы отдалять или приближать рабочее пространство и не взирая на количество таблиц комфортно работать с ними. Их можно увидеть на рисунке 11.

В центре располагается рабочее пространство, которое можно увидеть на рисунке 12.

Рисунок 12 – Прототип рабочего пространства

Наверху располагается Header на котором находятся иконки выполняющие различные функции. Его можно увидеть на рисунке 13.

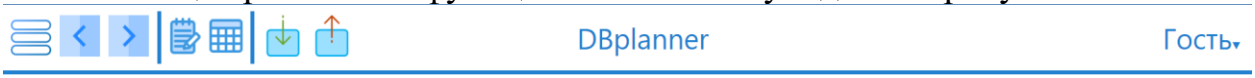


Рисунок 13 – Прототип header

Слева располагается иконка открывающегося меню. Смотрите рисунок 14.

Рисунок 14 – Прототип иконки открывающегося меню

В открывающемся меню, которое вы можете увидеть на рисунке 15, находятся три пункта создать, загрузить и сохранить.

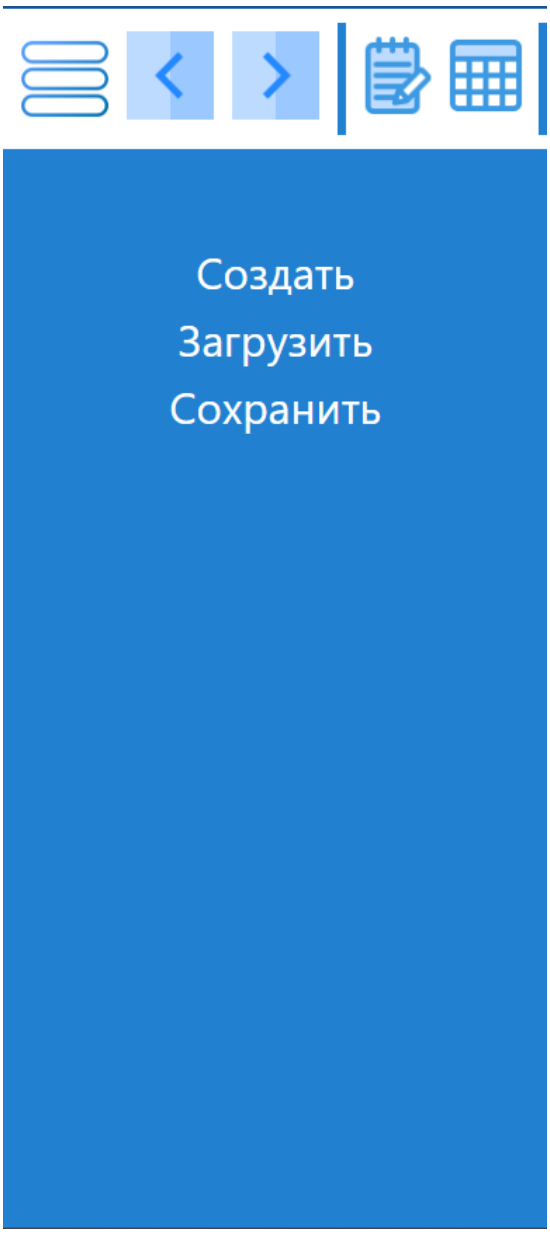


Рисунок 15 – Прототип открывающегося меню

Правее находятся иконки для отмены действий и перемещения между ними. Их вы можете увидеть на рисунке 16.

Рисунок 16 – Прототип иконок действий

Правее находятся иконки для вызова стикера для заметок и вызов окна создания таблицы. Они находятся на рисунке 17.

Рисунок 17 – Прототип иконок заметок и таблицы

На стикере мы можем написать какую-то информацию необходимую для нас. Это можно увидеть на рисунке 18.

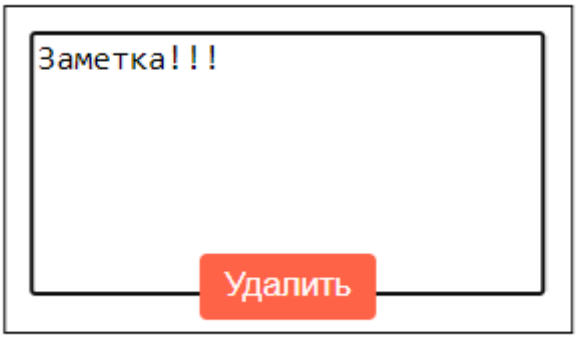


Рисунок 18 – Прототип стикера с заметками

На прототипе окна для создания таблиц, который можно посмотреть на рисунке 19, мы можем ввести название и выбрать тип данных, основной ключ, уникальное поле, автоинкремент и выбрать внешний ключ.

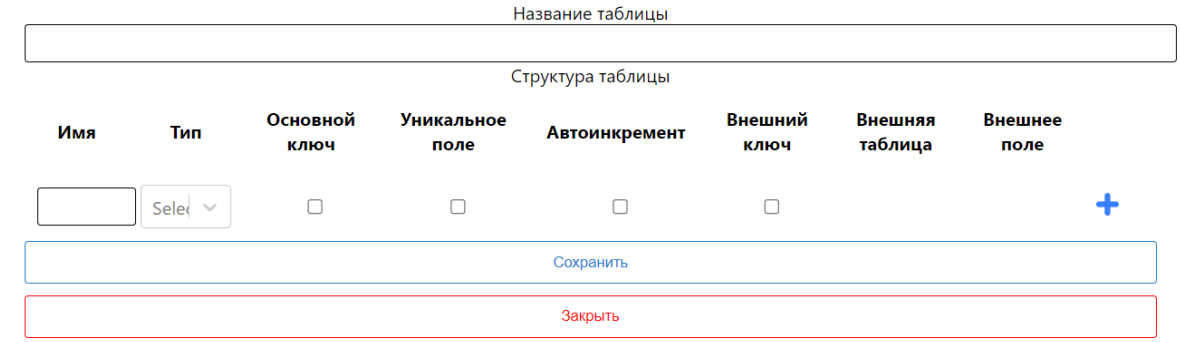


Рисунок 19 – Прототип окна создания таблицы

Далее идут иконки импорта и экспорта. Вы можете увидеть их на рисунке 20.

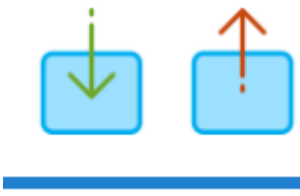


Рисунок 20 – Иконки импорта и экспорта.

В окне импорта, которое можно посмотреть на рисунке 21, мы можем выбрать тип SQL и сгенерировать скрипт.

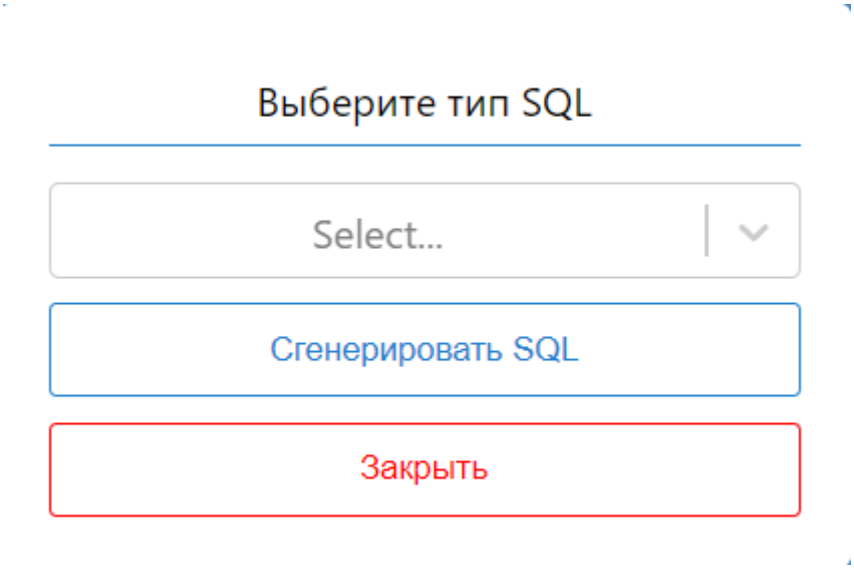


Рисунок 21 – Окно экспорта.

В центре находится название нашего приложения.

Рисунок 22 – Название приложения

В крайнем правом углу находится система регистрации

Рисунок 23 – Система регистрации до входа

После наведения на надпись гость нам откроется два варианта входа и регистрации

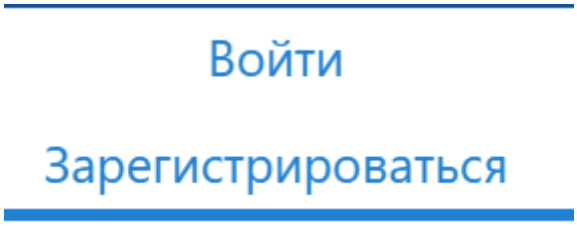


Рисунок 24 – Система авторизации

После выбора одного из двух вариантов нам откроется окно входа и регистрации соответственно. Смотрите на рисунках 25 и 26 соотеветсвенно.

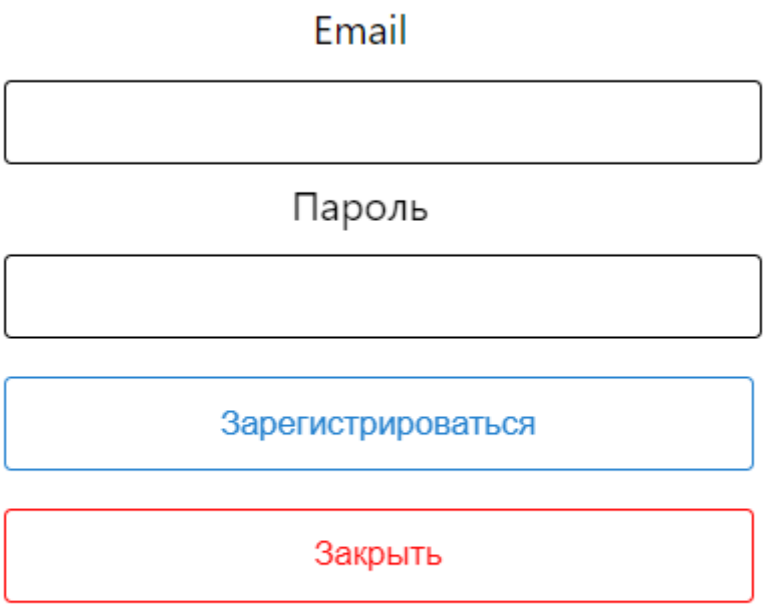


Рисунок 25 – Окно регистрации

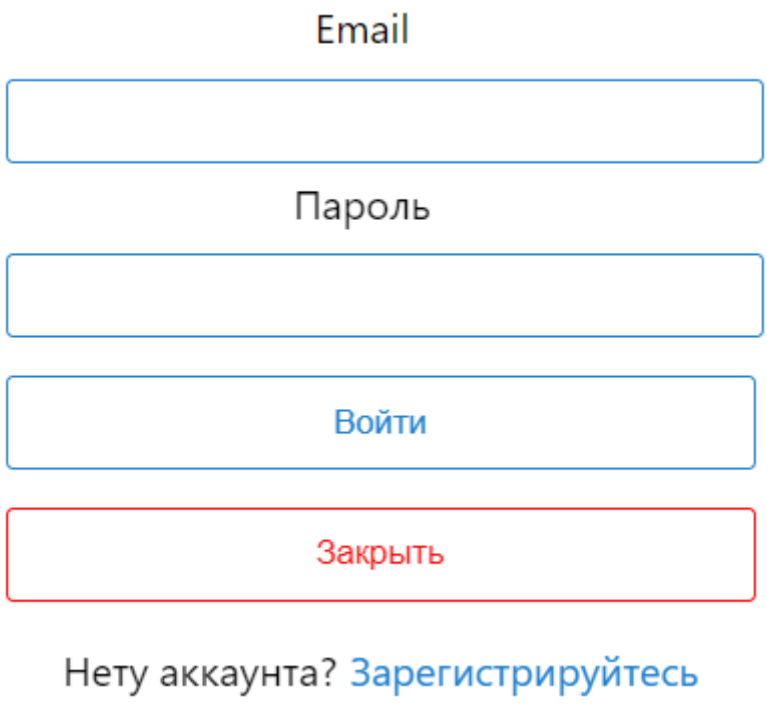


Рисунок 26 – Окно входа

6 Создание программного средства

6.1 Структура программы

Наше приложение представляет собой современное веб-приложение для управления задачами с использованием технологий Node.js, Express.js, React.js и PostgreSQL. Оно разделено на серверную и клиентскую части, каждая из которых имеет свою структуру и ответственности. Смотрите рисунок 27.

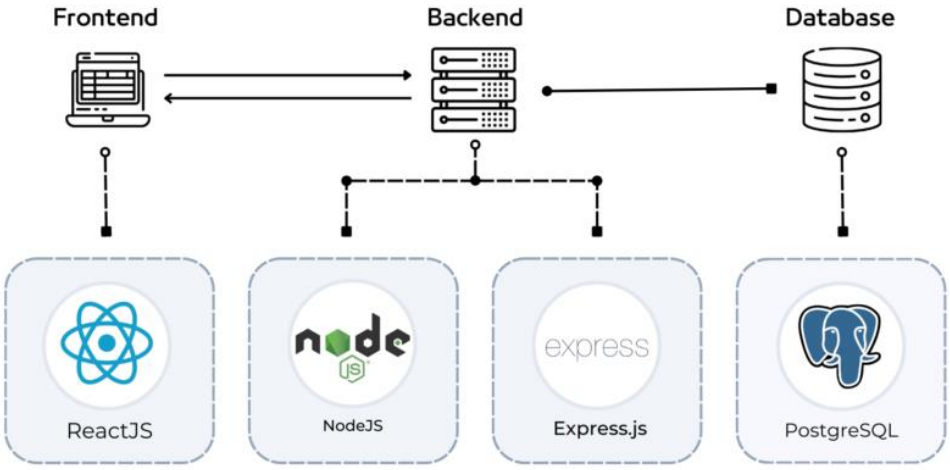


Рисунок 27 – Структура приложения

Рассмотрим отдельные подсистемы данной программы

1. Подсистема управления данными:
   - Включает в себя все функции, связанные с получением, обработкой, изменением и хранением данных, таких как таблицы, записи и запросы к базе данных.
2. Подсистема пользовательского интерфейса (UI):
   - Отвечает за отображение данных и взаимодействие с пользователем через интерфейс пользователя. Это включает в себя все компоненты и элементы управления, такие как кнопки, формы, таблицы и т. д.
3. Подсистема аутентификации и авторизации:
   - Отвечает за аутентификацию пользователей (проверка их личности) и авторизацию (управление правами доступа к различным ресурсам в приложении).
4. Подсистема обработки данных:
   - Включает в себя все функции и алгоритмы обработки данных, такие как фильтрация, сортировка, преобразование и т. д.
5. Подсистема экспорта и импорта данных:
   - Отвечает за возможность экспорта данных из приложения во внешние форматы и импорта данных из внешних источников.
6. Подсистема управления приложением:

- Включает в себя все функции, связанные с управлением состоянием приложения, маршрутизацией, модулями и компонентами.

7. Подсистема безопасности:
    - Отвечает за обеспечение безопасности приложения, включая защиту от угроз, аутентификацию пользователей, контроль доступа и шифрование данных.
8. Подсистема взаимодействия с внешними сервисами и API:
    - Отвечает за взаимодействие с внешними сервисами, API и другими компонентами, которые могут предоставлять дополнительную функциональность или данные для приложения.

6.2 Особенности используемых технологий

React.js — это библиотека JavaScript для создания пользовательских интерфейсов, разработанная компанией Facebook. Она предоставляет эффективные инструменты для создания интерактивных и масштабируемых веб-приложений. Основные принципы работы React.js включают в себя:

1. Компонентный подход: Основным строительным блоком React.js являются компоненты. Компоненты позволяют разбить пользовательский интерфейс на маленькие и независимые части, каждая из которых отвечает за определенный аспект поведения или отображения. Компоненты могут быть вложенными, что позволяет создавать сложные интерфейсы из простых компонентов.
2. Виртуальный DOM: React использует виртуальное представление DOM для эффективного обновления пользовательского интерфейса. Вместо манипуляций с реальным DOM напрямую, React создает в памяти виртуальное дерево DOM, которое затем сравнивается с реальным DOM и применяются только изменения, минимизируя количество операций на реальном DOM.
3. Однонаправленный поток данных (Unidirectional Data Flow): В React.js данные передаются вниз по иерархии компонентов. Это означает, что изменения данных в родительском компоненте автоматически приводят к обновлению дочерних компонентов. Этот принцип делает код более предсказуемым и управляемым.
4. Использование JSX: JSX (JavaScript XML) - это расширение языка JavaScript, которое позволяет писать HTML-подобный код внутри JavaScript. JSX делает код React более читабельным и легким для понимания, а также обеспечивает статическую типизацию и безопасность на этапе компиляции.
5. Композиция и переиспользование: Благодаря компонентному подходу React, компоненты могут быть переиспользованы в различных частях приложения. Это способствует созданию модульного и масштабируемого кода, а также упрощает его поддержку и обновление.

6. Жизненный цикл компонентов: Компоненты React имеют жизненный цикл, состоящий из различных методов, которые вызываются на разных этапах их жизни. Это позволяет разработчикам выполнять определенные действия при монтировании, обновлении или размонтировании компонентов, например, загрузка данных с сервера или выполнение очистки ресурсов.
7. Использование состояния (State): Состояние позволяет компонентам React хранить и управлять своим внутренним состоянием. Когда состояние компонента изменяется, React обновляет соответствующий пользовательский интерфейс, обеспечивая реактивное поведение приложения.
8. Эти принципы делают React.js мощным и эффективным инструментом для разработки современных веб-приложений, позволяя разработчикам создавать интерфейсы, которые легко поддерживать, масштабировать и расширять.

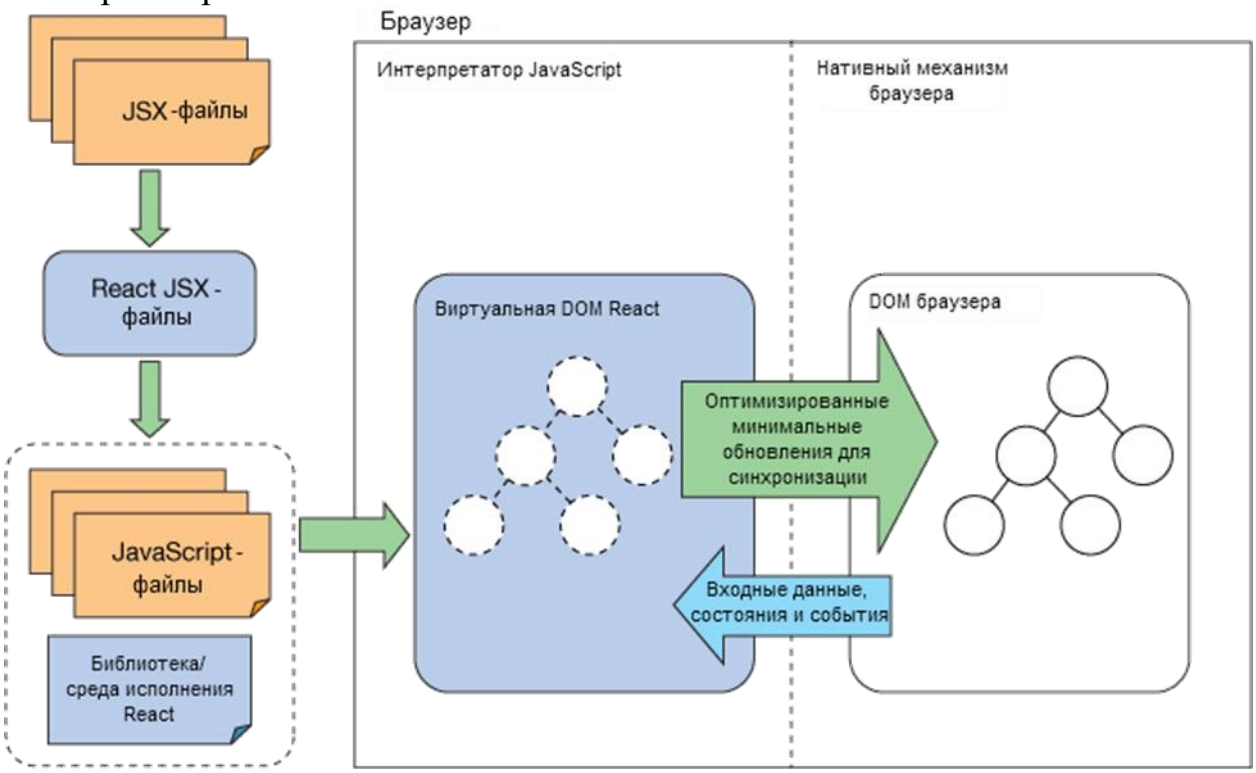


Рисунок 28 – Устройство React

Node.js и Express.js — это два популярных инструмента для разработки серверной части приложений на JavaScript. Давайте начнем с описания каждого из них и затем рассмотрим, как они взаимодействуют друг с другом.

Node.js — это среда выполнения JavaScript, построенная на движке V8 от Google Chrome. Она позволяет разработчикам запускать JavaScript на сервере вместо традиционного исполнения в браузере. Основные особенности Node.js включают в себя:

- Асинхронная и событийно-ориентированная архитектура: Node.js использует асинхронные операции ввода/вывода и событийно-

ориентированную архитектуру для обеспечения высокой производительности и масштабируемости приложений.

- Непрерывная работа: Node.js предназначен для работы в режиме непрерывной работы, что означает, что приложения могут обрабатывать большое количество одновременных подключений без блокировки потока.
- Модульная система: Node.js использует модульную систему CommonJS для организации кода. Это позволяет разработчикам разделять код на небольшие модули и повторно использовать их в разных частях приложения.

Express.js — это минималистичный и гибкий веб-фреймворк для Node.js. Он облегчает создание веб-приложений и API, предоставляя множество полезных функций и утилит. Основные черты Express.js:

- Асинхронная и событийно-ориентированная архитектура: Node.js использует асинхронные операции ввода/вывода и событийно-ориентированную архитектуру для обеспечения высокой производительности и масштабируемости приложений.
- Непрерывная работа: Node.js предназначен для работы в режиме непрерывной работы, что означает, что приложения могут обрабатывать большое количество одновременных подключений без блокировки потока.
- Модульная система: Node.js использует модульную систему CommonJS для организации кода. Это позволяет разработчикам разделять код на небольшие модули и повторно использовать их в разных частях приложения.

Как они работают вместе:

Express.js обычно используется как фреймворк для создания веб-приложений на базе Node.js. Вы определяете ваше приложение Express.js, определяя маршруты, обработчики и используя промежуточные функции, и затем запускаете его на сервере Node.js.

Node.js, в свою очередь, является средой выполнения, которая позволяет запускать ваше веб-приложение Express.js. Он обрабатывает все входящие запросы, передавая их вашему приложению Express.js для обработки, и отправляет обратно соответствующие HTTP-ответы.

Таким образом, Express.js является инструментом для облегчения разработки веб-приложений на базе Node.js, предоставляя удобный интерфейс и множество полезных функций для работы с HTTP-запросами и ответами.

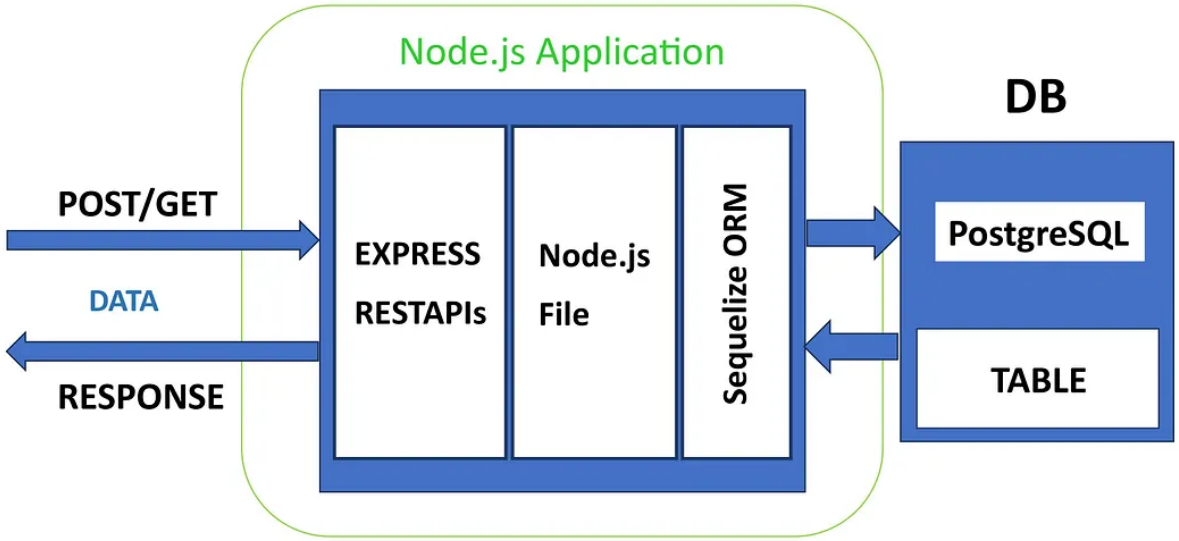


Рисунок 29 – Устройство Node js

Вот описание основных принципов работы PostgreSQL:

1. Реляционная модель данных: PostgreSQL основан на реляционной модели данных, что означает, что данные организованы в таблицы с определенными отношениями между ними. Это позволяет эффективно хранить и управлять структурированными данными, такими как пользователи, заказы или товары.
2. ACID-совместимость: PostgreSQL гарантирует ACID-свойства для транзакций, что обеспечивает надежность и целостность данных. ACID означает атомарность (Atomicity), согласованность (Consistency), изолированность (Isolation) и долговечность (Durability), что делает PostgreSQL привлекательным выбором для приложений, требующих высокой надежности и безопасности данных.
3. Многопользовательский доступ**:** PostgreSQL поддерживает многопользовательский доступ к базе данных, что позволяет нескольким пользователям одновременно работать с данными. Благодаря механизмам управления правами доступа, аутентификации и авторизации, PostgreSQL обеспечивает безопасность данных и контроль над доступом к ним.
4. Мощный язык запросов: PostgreSQL предоставляет мощный язык запросов SQL для манипуляции данными в базе данных. Он поддерживает широкий спектр операторов, функций и агрегатных функций, что делает возможным выполнение сложных запросов для извлечения, изменения и анализа данных.

5. Индексы и оптимизация запросов: PostgreSQL поддерживает создание индексов, которые ускоряют выполнение запросов к базе данных. Также он предоставляет механизмы для оптимизации запросов, такие как анализатор запросов, планировщик запросов и оптимизатор запросов, которые помогают обеспечить эффективное выполнение запросов даже на больших объемах данных.
6. Расширяемость: PostgreSQL предоставляет механизмы для расширения функциональности базы данных путем создания пользовательских типов данных, функций и операторов. Это позволяет разработчикам создавать собственные расширения и модули для удовлетворения специфических потребностей и требований их приложений.
7. Открытый исходный код и активное сообщество: PostgreSQL является проектом с открытым исходным кодом, что означает, что его исходный код доступен для свободного использования, модификации и распространения. Он имеет активное сообщество разработчиков, которые постоянно работают над улучшением и развитием базы данных, обеспечивая поддержку и обновления на протяжении многих лет.

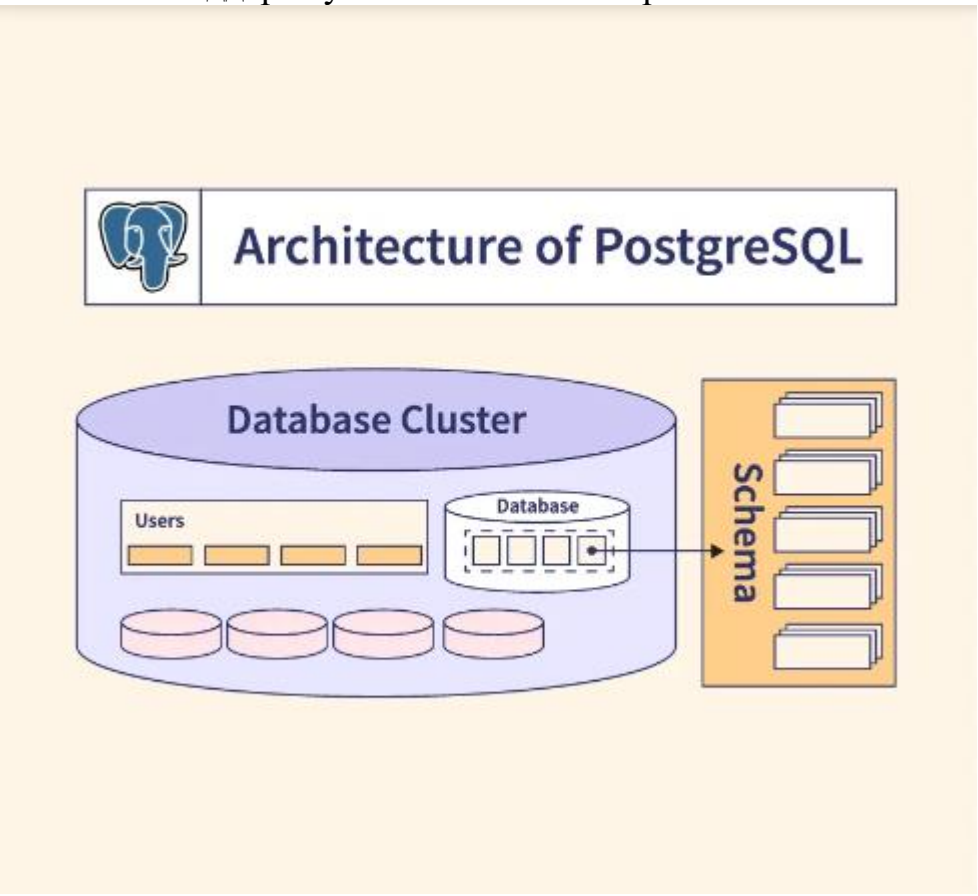


Рисунок 30 – Архитектура PostgreSQL

6.3 Используемы библиотеки

axios — это библиотека JavaScript для выполнения HTTP запросов из браузера или из Node.js. Она предоставляет простой и удобный интерфейс для взаимодействия с внешними API, отправки данных на сервер и получения ответов.

Вот основные характеристики и функции библиотеки axios:

1. Простота использования: axios предоставляет простой API для отправки HTTP запросов и обработки ответов.
2. Поддержка промисов: axios основан на промисах, что делает его легким в использовании с async/await и обеспечивает удобную обработку асинхронных запросов.
3. Кросс-платформенность: axios может использоваться как в браузере, так и в Node.js, что делает его универсальным инструментом для работы с HTTP запросами в различных окружениях.
4. Поддержка интерсепторов: axios позволяет добавлять промежуточные обработчики для запросов и ответов, что облегчает выполнение различных действий, таких как авторизация, обработка ошибок и многое другое.
5. Поддержка заголовков и параметров запроса: axios позволяет устанавливать заголовки запросов и передавать параметры через URL строки или тело запроса.

react-select — это библиотека для React, которая предоставляет гибкий и удобный компонент выпадающего списка. Она позволяет создавать красивые и функциональные выпадающие списки с поддержкой различных функций, таких как поиск, выбор множественных или одиночных значений, настраиваемый внешний вид и многое другое.

Вот основные характеристики и функции библиотеки react-select:

1. Поиск: react-select позволяет выполнять поиск по элементам списка, что делает его удобным для использования в списках с большим количеством элементов.
2. Множественный выбор: Компонент поддерживает как одиночный, так и множественный выбор значений, что позволяет пользователям выбирать несколько элементов из списка.
3. Настройка внешнего вида: react-select обеспечивает гибкую настройку внешнего вида компонента, что позволяет легко интегрировать его в дизайн вашего приложения.
4. Кастомизация: Вы можете легко кастомизировать стили и поведение react-select, добавляя собственные компоненты, функции фильтрации и многое другое.
5. Поддержка различных типов данных: Компонент react-select может работать с различными типами данных, включая массивы объектов, строки, числа и т. д.

6. Поддержка клавиатуры: react-select поддерживает навигацию с клавиатуры, что делает его удобным в использовании для пользователей с ограниченными возможностями.

Библиотека cors (Cross-Origin Resource Sharing) представляет собой инструмент для управления политикой совместного использования ресурсов между различными источниками (origin) веб-приложений. Она используется в серверных приложениях для обеспечения безопасности и разрешения запросов от клиентов, которые происходят с других источников, доменов или портов.

Вот основные аспекты и функциональные возможности библиотеки cors:

1. Разрешение CORS-запросов: cors позволяет серверу обрабатывать запросы от клиентов, которые происходят из других источников (origin), чем серверный домен. Это включает запросы из разных доменов, поддоменов, протоколов или портов.
2. Конфигурация политики CORS: Библиотека позволяет настраивать различные аспекты политики CORS, такие как допустимые источники (origin), методы запросов (GET, POST, PUT, DELETE и т. д.), заголовки запросов и разрешение куки.
3. Простота в использовании: cors обеспечивает простой и интуитивно понятный интерфейс для установки и настройки политики CORS на сервере. Он предлагает гибкие параметры конфигурации, чтобы вы могли точно определить, какие запросы разрешены и какие должны быть отклонены.
4. Защита от CSRF-атак: Использование политики CORS помогает защитить ваше веб-приложение от атак типа Cross-Site Request Forgery (CSRF), так как браузер будет блокировать запросы, которые не соответствуют установленным правилам CORS.

react-router-dom — это библиотека для маршрутизации веб-приложений на основе React. Она позволяет создавать одностраничные приложения (SPA), где контент меняется динамически без перезагрузки страницы.

Вот основные аспекты и функциональные возможности react-router-dom:

1. BrowserRouter: Этот компонент используется для обертывания всего приложения и создания контекста маршрутизации. Он использует HTML5 History API для манипуляции URL-адресом в адресной строке.
2. Route: Компонент Route используется для определения соответствия пути URL с компонентом React. Когда адрес в адресной строке совпадает с указанным путем, Route рендерит соответствующий компонент.
3. Switch: Switch используется для обертывания компонентов Route и гарантирует, что будет отображен только один первый подходящий

маршрут. Это полезно, когда необходимо избежать множественного сопоставления.

4. Link и NavLink: Компоненты Link и NavLink используются для создания ссылок между различными маршрутами в вашем приложении. Они позволяют переходить между маршрутами без перезагрузки страницы.
5. Redirect: Redirect используется для перенаправления пользователя на другой маршрут. Он может быть использован программно или в ответ на определенные условия или действия пользователя.

6.4 Хранимые состояния

Хуки (Hooks) в React — это нововведение, представленное в React 16.8, которое позволяет использовать состояние и другие возможности React без необходимости создавать классовые компоненты. Они обеспечивают более простой и гибкий способ написания компонентов, особенно функциональных.

useState — это хук, предоставляемый библиотекой React, который позволяет функциональным компонентам хранить внутреннее состояние. Этот хук решает проблему отсутствия состояния в функциональных компонентах, делая их более мощными и гибкими.

Основные особенности useState:

1. Хранение состояния: С помощью useState компоненты могут хранить переменные, которые изменяются в течение их жизненного цикла. Каждый вызов компонента создает свой собственный экземпляр состояния.
2. Обновление состояния: useState возвращает пару значений: текущее состояние и функцию для его обновления. Обновление состояния может происходить асинхронно и может зависеть от предыдущего состояния.
3. Неизменяемость состояния: React гарантирует, что при обновлении состояния предыдущее состояние не изменяется напрямую. Вместо этого используется новое значение состояния, возвращаемое функцией обновления.

useEffect — это хук, предоставляемый библиотекой React, который позволяет выполнять побочные эффекты в функциональных компонентах. Побочные эффекты включают в себя такие операции, как загрузка данных из сети, обработка событий и изменение DOM. useEffect позволяет управлять жизненным циклом компонента, аналогично методам жизненного цикла в классовых компонентах.

Основные особенности useEffect:

1. Выполнение побочных эффектов: useEffect позволяет выполнять код, который должен быть выполнен после рендеринга компонента. Этот код может быть асинхронным и не блокирует интерфейс.
2. Очистка эффектов: useEffect возвращает функцию, которая может быть использована для очистки или отмены побочных эффектов. Это

особенно полезно для выполнения уборки перед удалением компонента из DOM или для отмены подписок.

3. Управление зависимостями: useEffect может принимать второй аргумент - массив зависимостей. Если зависимости изменяются между рендерами компонента, useEffect будет запущен снова. Это позволяет оптимизировать производительность и избегать лишних запусков.

В контексте React, состояние (state) представляет собой данные, которые управляются внутри компонентов. Состояние определяет, как компонент выглядит и ведет себя в определенный момент времени, и может изменяться в процессе жизненного цикла компонента.

Основные характеристики состояния:

1. Локальность: Состояние управляется локально в пределах компонента. Это означает, что каждый компонент может иметь свое собственное состояние, и изменения в одном компоненте не затрагивают состояние других компонентов.
2. Изменяемость: Состояние в React является изменяемым. Это означает, что его можно обновлять в течение жизненного цикла компонента с использованием специальных функций, предоставляемых React, таких как setState или хуки состояния, например, useState.
3. Обновление и перерендеринг: Когда состояние компонента изменяется, React обновляет визуальное представление компонента, чтобы отразить новое состояние. Это процесс, известный как перерендеринг, и он происходит автоматически.
4. Инициализация: Начальное значение состояния обычно устанавливается в методе constructor (для классовых компонентов) или с помощью хуков состояния (для функциональных компонентов).
5. Пропаганда вниз (Downward Data Flow): В React данные передаются от родительских компонентов к дочерним через пропсы. Состояние обычно управляется в родительских компонентах и передается дочерним компонентам в виде пропсов.

Состояние, которые содержатся в данной программе:

```
const [tableData, setTableData] = useState([]);
```

Это состояние представляет собой переменную tableData, которая хранит данные о таблицах внутри компонента React. В данном случае, она инициализируется пустым массивом [], что означает, что изначально данных о таблицах нет.

Помимо этого, используется функция setTableData, которая предоставляется React и позволяет обновлять значение переменной tableData. Когда вызывается setTableData с новым значением, React автоматически перерендеривает компонент, чтобы отобразить изменения.

Таким образом, это состояние tableData используется для хранения и управления данными о таблицах внутри компонентов.

```
const [isGuestMenuOpen, setGuestMenuOpen] = useState(false);
```

Это состояние isGuestMenuOpen представляет собой булевое значение, которое определяет, открыто ли гостевое меню в интерфейсе. Изначально оно устанавливается в false, что означает, что гостевое меню закрыто.

Функция setGuestMenuOpen позволяет изменять это состояние. При вызове этой функции с аргументом true, гостевое меню открывается, а при вызове с аргументом false, оно закрывается.

Таким образом, это состояние используется для управления видимостью гостевого меню в интерфейсе.

```
const [isBarMenuOpen, setBarMenuOpen] = useState(false);
```

Это состояние isBarMenuOpen представляет собой булевое значение, которое определяет, открыто ли боковое меню в интерфейсе. Изначально оно устанавливается в false, что означает, что боковое меню закрыто.

Функция setBarMenuOpen позволяет изменять это состояние. При вызове этой функции с аргументом true, боковое меню открывается, а при вызове с аргументом false, оно закрывается.

Таким образом, это состояние используется для управления видимостью бокового меню в интерфейсе.

```
const [stickers, setStickers] = useState([]);
```

Это состояние stickers представляет собой массив, который хранит информацию о наклейках (stickers) в приложении. Изначально массив пустой.

Функция setStickers используется для изменения этого состояния. При вызове функции setStickers с новым значением, состояние stickers обновляется, а компонент, использующий это состояние, перерендеривается с новым списком наклеек.

Таким образом, это состояние используется для управления и отображения наклеек в пользовательском интерфейсе приложения.

```
const [isCreateOpen, setIsCreateOpen] = useState(false);
```

Это состояние isCreateOpen используется для отслеживания состояния открытия или закрытия определенной области или компонента в приложении. Изначально установлено в значение false, что означает, что область или компонент закрыт.

Функция setIsCreateOpen предназначена для изменения этого состояния. Когда вызывается функция setIsCreateOpen(true), область или компонент становится видимыми (открытыми), а когда вызывается функция setIsCreateOpen(false), область или компонент скрывается (закрывается).

Это состояние может использоваться, например, для управления отображением модального окна, выпадающего меню или любого другого компонента, который может быть открыт или закрыт в зависимости от действий пользователя или состояния приложения.

```
const [isExportOpen, setExportOpen] = useState(false);
```

Это состояние isExportOpen используется для отслеживания состояния открытия или закрытия области или компонента, связанного с экспортом данных в приложении. Изначально установлено в значение false, что означает, что область или компонент, связанный с экспортом данных, закрыт.

Функция setExportOpen предназначена для изменения этого состояния. Когда вызывается функция setExportOpen(true), область или компонент, связанный с экспортом данных, становится видимыми (открытыми), а когда вызывается функция setExportOpen(false), область или компонент скрывается (закрывается).

Это состояние может использоваться, например, для управления отображением модального окна или панели с опциями экспорта данных, которая может быть открыта или закрыта в зависимости от действий пользователя или состояния приложения.

```
const [isImportOpen, setImportOpen] = useState(false);
```

Это состояние isImportOpen используется для отслеживания состояния открытия или закрытия определенного компонента, связанного с импортом данных, в приложении. Изначально установлено в значение false, что означает, что компонент, связанный с импортом данных закрыт.

Функция setImportOpen предназначена для изменения этого состояния. Когда вызывается функция setImportOpen(true), компонент, связанный с импортом данных, становится видимым (открытым), а когда вызывается функция setImportOpen(false), компонент скрывается (закрывается).

Это состояние может использоваться, например, для управления отображением компонента импорта данных, который может быть открыт или закрыт в зависимости от действий пользователя или состояния приложения.

```
const [isRegOpen, setRegOpen] = useState(false);
```

Это состояние isRegOpen используется для отслеживания состояния открытия или закрытия определенного компонента, связанного с регистрацией пользователей в приложении. Изначально установлено в значение false, что означает, что компонент, связанный с регистрацией, закрыт.

Функция setRegOpen предназначена для изменения этого состояния. Когда вызывается функция setRegOpen(true), компонент, связанный с регистрацией, становится видимым (открытым), а когда вызывается функция setRegOpen(false), компонент скрывается (закрывается).

Это состояние может использоваться, например, для управления отображением компонента регистрации пользователей, который может быть открыт или закрыт в зависимости от действий пользователя или состояния приложения.

```
const [isLoginOpen, setLoginOpen] = useState(false);
```

Это состояние isLoginOpen используется для отслеживания состояния открытия или закрытия компонента, связанного с процессом входа в систему. Изначально установлено в значение false, что означает, что компонент входа в систему закрыт.

Функция setLoginOpen предназначена для изменения этого состояния. Когда вызывается функция setLoginOpen(true), компонент входа в систему становится видимым (открытым), а когда вызывается функция setLoginOpen(false), компонент скрывается (закрывается).

Это состояние может использоваться для управления отображением компонента входа в систему, который может быть открыт или закрыт в зависимости от действий пользователя или состояния приложения.

```
const [isTableOpen, setTableOpen] = useState(false);
```

Это состояние isTableOpen используется для отслеживания состояния открытия или закрытия определенной таблицы или компонента, связанного с таблицей, в приложении. Изначально установлено в значение false, что означает, что таблица или компонент, связанный с таблицей, закрыт.

Функция setTableOpen предназначена для изменения этого состояния. Когда вызывается функция setTableOpen(true), таблица или компонент, связанный с таблицей, становится видимым (открытым), а когда вызывается функция setTableOpen(false), таблица или компонент скрывается (закрывается).

Это состояние может использоваться, например, для управления отображением таблицы или панели с данными, которая может быть открыта или закрыта в зависимости от действий пользователя или состояния приложения.

6.5 Отдельные функции

Popup (всплывающее окно) — это маленькое окно, которое внезапно появляется поверх текущего содержимого страницы. Оно обычно содержит важное сообщение, дополнительные настройки или действия для пользователя. Всплывающие окна используются для предоставления информации без переключения на другие страницы или открытия новых вкладок.

Основные характеристики всплывающих окон:

1. Появление поверх текущего контента: Всплывающее окно отображается поверх текущего содержимого страницы, часто с затемнением фона для привлечения внимания пользователя к его содержимому.
2. Временное или периодическое: Окно может быть временным, появляясь только при определенных условиях или действиях пользователя, или появляться периодически для отображения важных уведомлений.
3. Содержание и функциональность: Всплывающие окна могут содержать текст, изображения, формы для ввода данных, кнопки для выполнения действий и т. д. Они могут использоваться для запроса подтверждения, предупреждений, уведомлений или предложения дополнительных действий.
4. Взаимодействие с пользователем: Пользователь может закрыть всплывающее окно с помощью кнопки "закрыть" или щелчка вне окна. Некоторые всплывающие окна могут иметь определенное время жизни, после чего они автоматически закрываются.
5. Адаптивность и стиль: Всплывающие окна могут быть адаптивными и стилизованными в соответствии с общим дизайном и стилем приложения или web-приложения. Они могут быть адаптированы для

мобильных устройств и настольных компьютеров для обеспечения лучшего пользовательского опыта.

6. Примеры использования: Всплывающие окна используются для предупреждений о критических ошибках, запросов на подтверждение удаления данных, уведомлений о новых сообщениях или событиях, предложений о подписке на рассылку и многих других ситуациях, где важно привлечь внимание пользователя к определенной информации или действию.

```
const toggleTablePopup = () => {
        setTableOpen(true);
    }
```

Эта функция toggleTablePopup предназначена для открытия компонента, связанного с таблицей, путем установки состояния isTableOpen в true. Когда вызывается функция toggleTablePopup(), компонент становится видимым (открытым), что позволяет пользователю взаимодействовать с таблицей или просматривать ее содержимое.

```
const toggleCreatePopup = () => {
        setIsCreateOpen(true);
    };
```

Функция toggleCreatePopup предназначена для открытия компонента, связанного с созданием, путем установки состояния isCreateOpen в true. Когда вызывается функция toggleCreatePopup(), компонент становится видимым (открытым), что позволяет пользователю начать процесс создания нового элемента или объекта.

```
const toggleExportPopup = () => {
        setExportOpen(true);
    };
```

Функция toggleExportPopup используется для открытия компонента, связанного с экспортом данных. Она устанавливает состояние isExportOpen в true, что приводит к отображению (открытию) компонента экспорта данных. Это позволяет пользователю выбрать тип экспорта и сгенерировать соответствующий файл.

```
const toggleImportPopup = () => {
        setImportOpen(true);
    };
```

Функция toggleImportPopup предназначена для открытия всплывающего окна, которое, вероятно, используется для импорта данных или выполнения каких-либо операций, связанных с импортом. Вот как это работает:

1. toggleImportPopup: Это функция, которая вызывается при некотором событии, например, при щелчке на кнопке "Импорт" или при выполнении определенного действия. В данном контексте функция предназначена для открытия всплывающего окна.
2. setImportOpen(true): Эта часть кода использует функцию setImportOpen, возвращаемую хуком useState, чтобы изменить состояние переменной isImportOpen на значение true. Предполагается, что переменная

isImportOpen используется для определения состояния всплывающего окна. Когда isImportOpen равно true, всплывающее окно отображается на экране.

Таким образом, когда функция toggleImportPopup вызывается, она устанавливает переменную isImportOpen в значение true, что приводит к открытию всплывающего окна, связанного с импортом данных или другими операциями импорта.

```
const toggleRegPopup = () => {
        setRegOpen(true);
        setLoginOpen(false);
    };
```

Функция toggleRegPopup предназначена для открытия всплывающего окна, связанного с регистрацией пользователя, и одновременного закрытия всплывающего окна, связанного с входом пользователя. Вот как это работает:

1. setRegOpen(true): Этот код использует функцию setRegOpen, возвращаемую хуком useState, чтобы установить состояние переменной isRegOpen в значение true. Предполагается, что переменная isRegOpen используется для определения состояния всплывающего окна регистрации. Когда isRegOpen равно true, всплывающее окно регистрации отображается на экране.
2. setLoginOpen(false): Этот код использует функцию setLoginOpen, возвращаемую хуком useState, чтобы установить состояние переменной isLoginOpen в значение false. Предполагается, что переменная isLoginOpen также используется для определения состояния всплывающего окна входа. Установка isLoginOpen в значение false приводит к закрытию всплывающего окна входа.

Таким образом, когда функция toggleRegPopup вызывается, она открывает всплывающее окно регистрации, устанавливая переменную isRegOpen в значение true, и одновременно закрывает всплывающее окно входа, устанавливая переменную isLoginOpen в значение false.

```
const toggleLoginPopup = () => {
        setLoginOpen(true);
        setRegOpen(false);
    };
```

Функция toggleLoginPopup предназначена для открытия всплывающего окна, связанного с входом пользователя, и одновременного закрытия всплывающего окна, связанного с регистрацией пользователя. Вот как это работает:

1. setLoginOpen(true): Этот код использует функцию setLoginOpen, возвращаемую хуком useState, чтобы установить состояние переменной isLoginOpen в значение true. Предполагается, что переменная isLoginOpen используется для определения состояния всплывающего

окна входа. Когда isLoginOpen равно true, всплывающее окно входа отображается на экране.

2. setRegOpen(false): Этот код использует функцию setRegOpen, возвращаемую хуком useState, чтобы установить состояние переменной isRegOpen в значение false. Предполагается, что переменная isRegOpen используется для определения состояния всплывающего окна регистрации. Установка isRegOpen в значение false приводит к закрытию всплывающего окна регистрации.

Таким образом, когда функция toggleLoginPopup вызывается, она открывает всплывающее окно входа, устанавливая переменную isLoginOpen в значение true, и одновременно закрывает всплывающее окно регистрации, устанавливая переменную isRegOpen в значение false.

```
const toggleGuestMenu = () => {
        setGuestMenuOpen(!isGuestMenuOpen);
    };
```

Функция toggleGuestMenu предназначена для изменения состояния переменной isGuestMenuOpen, которая определяет, открыто ли меню для гостя. Вот как она работает:

1. setGuestMenuOpen(!isGuestMenuOpen): Этот код использует функцию setGuestMenuOpen, возвращаемую хуком useState, чтобы установить состояние переменной isGuestMenuOpen в противоположное от текущего. То есть, если isGuestMenuOpen равно true, после выполнения этой операции оно станет false, и наоборот.

Таким образом, при каждом вызове функции toggleGuestMenu происходит изменение состояния переменной isGuestMenuOpen с открытого на закрытое и наоборот. Это позволяет контролировать отображение и скрытие меню для гостя при взаимодействии с интерфейсом приложения.

```
const toggleBarMenu = () => {
        setBarMenuOpen(!isBarMenuOpen);
    };
```

Функция toggleBarMenu предназначена для изменения состояния переменной isBarMenuOpen, которая определяет, открыто ли боковое меню. Вот как она работает:

1. setBarMenuOpen(!isBarMenuOpen): Этот код использует функцию setBarMenuOpen, возвращаемую хуком useState, чтобы установить состояние переменной isBarMenuOpen в противоположное от текущего. То есть, если isBarMenuOpen равно true, после выполнения этой операции оно станет false, и наоборот.

Таким образом, при каждом вызове функции toggleBarMenu происходит изменение состояния переменной isBarMenuOpen с открытого на закрытое и наоборот. Это позволяет контролировать отображение и скрытие бокового меню при взаимодействии с интерфейсом приложения.

```
const handleIconClick = () => {
        const newStickers = [...stickers, <Sticker key={stickers.length}
onDelete={() => handleStickerDelete(stickers.length)} />];
        setStickers(newStickers);
    };
```

Эта функция handleIconClick вызывается при щелчке по иконке или другому элементу интерфейса, и она выполняет следующие действия:

1. Создание нового массива newStickers: В этой строке создается новый массив newStickers, который копирует все элементы из текущего массива stickers с помощью оператора распыления ...stickers. После этого добавляется новый элемент - компонент <Sticker />, который имеет ключ, равный текущей длине массива stickers. Ключи используются React для оптимизации производительности и корректного обновления элементов при их изменении.
2. Обновление состояния stickers: После создания нового массива newStickers он устанавливается в качестве нового значения для состояния stickers с помощью функции setStickers(newStickers). Это приводит к перерисовке компонента, так как React обнаруживает изменение в состоянии, и отображение новых стикеров на экране.

```
const handleStickerDelete = (index) => {
        const updatedStickers = stickers.filter((sticker, i) => i !== index);
        setStickers(updatedStickers);
    };
```

Эта функция handleStickerDelete вызывается при удалении стикера и принимает индекс удаляемого стикера в массиве stickers. Вот что она делает:

1. Фильтрация стикеров: В этой строке используется метод массива filter, чтобы создать новый массив updatedStickers, который содержит все стикеры, кроме того, который нужно удалить. Мы проходим по каждому стикеру в массиве stickers, и если его индекс i не совпадает с индексом удаляемого стикера index, то этот стикер остается в новом массиве updatedStickers.
2. Обновление состояния stickers: После того как массив updatedStickers сформирован, он устанавливается в качестве нового значения для состояния stickers с помощью функции setStickers(updatedStickers). Это приводит к перерисовке компонента, так как React обнаруживает изменение в состоянии, и отображение обновленного списка стикеров на экране.

6.6 Связь модулей

В контексте React.js термины "page" и "components" обычно используются для организации пользовательского интерфейса и структурирования кода приложения.

1. Page (Страница):

- В React.js "страница" обычно представляет собой компонент, который отображает содержимое конкретной страницы вашего веб-приложения. Каждая страница может содержать уникальный URL, различное содержимое и логику, и может быть отображена в браузере как отдельная веб-страница.
- Страницы могут быть организованы в иерархию и включать в себя другие компоненты, такие как заголовки, навигационные панели, контент и т. д.
- Обычно в React.js каждая страница представлена отдельным компонентом, который может быть маршрутизирован с помощью библиотеки маршрутизации, такой как React Router.

2. Component (Компонент):
   - В React.js "компонент" представляет собой многократно используемый строительный блок интерфейса, который объединяет в себе HTML, CSS и JavaScript для создания отдельных частей пользовательского интерфейса.
   - Компоненты могут быть функциональными или классовыми. Функциональные компоненты обычно определяются как функции JavaScript, возвращающие JSX (расширение языка JavaScript, позволяющее писать HTML внутри JavaScript). Классовые компоненты наследуют от базового класса React.Component.
   - Компоненты могут принимать входные данные, называемые "props" (свойства), и возвращать иерархию React-элементов, которые описывают структуру интерфейса. Компоненты также могут иметь состояние, управляемое React, с помощью хука useState или метода setState.
   - Компоненты могут быть вложенными друг в друга, что позволяет создавать модульный и масштабируемый код приложения.

Компоненты в React.js обеспечивают модульность, переиспользование кода и удобство разработки интерфейса, а страницы позволяют организовать пользовательский интерфейс приложения на уровне маршрутизации и навигации.

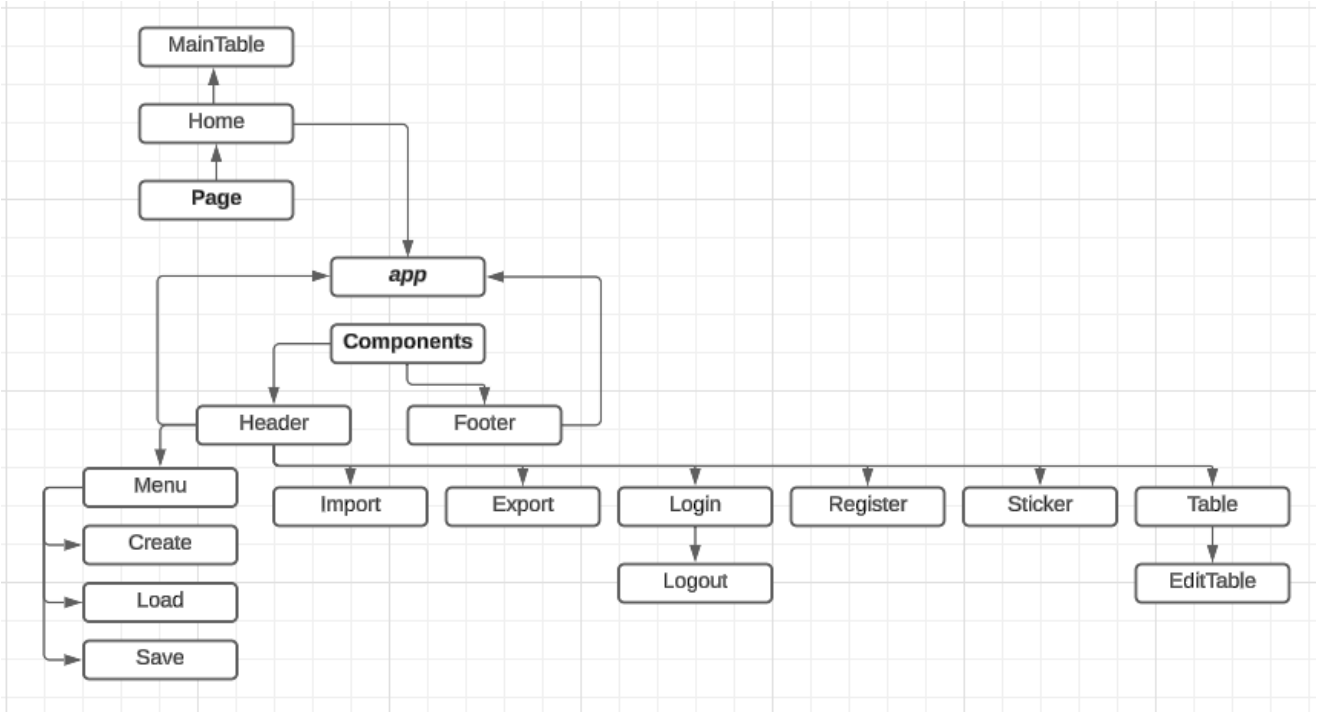


Рисунок 31 – Связь модулей

Далее разберем функцию всех модулей. Модули можно увидеть на рисунке 31, а разбор их функций в таблице 3.

Таблица 3 – Функции модулей

Название	Функция	Тип
Header	Header компонент предназначен для отображения верхней приложения. Он содержит элементы, такие как название, навигационное меню, ссылки на основные разделы приложения.	Component
Menu	Навигационное меню — это часть веб-интерфейса, предназначенная для навигации пользователя по различным разделам приложения. Содержит функциональные возможности, которые предоставляются ниже.	
Create	Компонент, где мы можем ввести название схемы.	
Load	Компонент для загрузки одной из схем (в случае если пользователь авторизован)	
Save	Компонент для сохранения изменений в схеме	
Import	Компонент для импорта схемы	
Export	Компонент для экспорта схемы	
Login	Компонент для входа	
Logout	Компонент для выхода	
Register	Компонент для регистрации	
Sticker	Компонент для размещения заметок	

Table	Компонент для создания таблицы	
EditTable	Компонент для редактирования таблицы	
MainTable	Компонент для отображения таблицы на рабочем пространстве	
Home	Главная страница	Page

7 Тестирование

Тестирование программы — это процесс оценки программного обеспечения с целью выявления и устранения дефектов, а также подтверждения того, что оно соответствует заявленным требованиям.

В тестирование мы проверим все варианты генерации скрипта, с авторизацией и без, и со всеми типами SQL.

Таблица 4 – Тестирование

	Тест 1	Тест 2	Тест 3	Тест 4
Oracle	+	-	-	-
MS SQL	-	+	-	-
My SQL	-	-	+	-
PostgreSQL	-	-	-	+
Результат	Успех	Успех	Успех	Успех

Как мы видим, в таблице 4 представлены результаты тестирования. Эти результаты говорят нам о корректной работе веб-приложения во всех сценариях.

Создадим теоретическую схему с большим количеством таблиц и внешних ключей дабы проиллюстрировать работоспособность приложения.

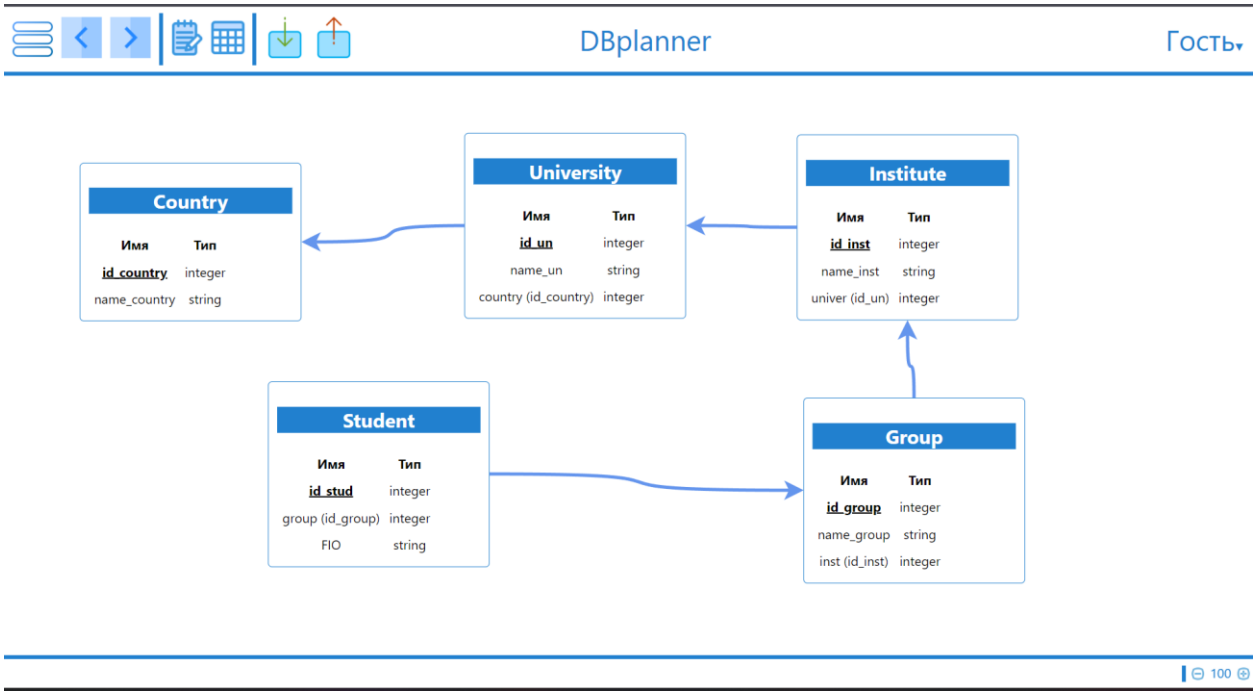


Рисунок 32 – Схема

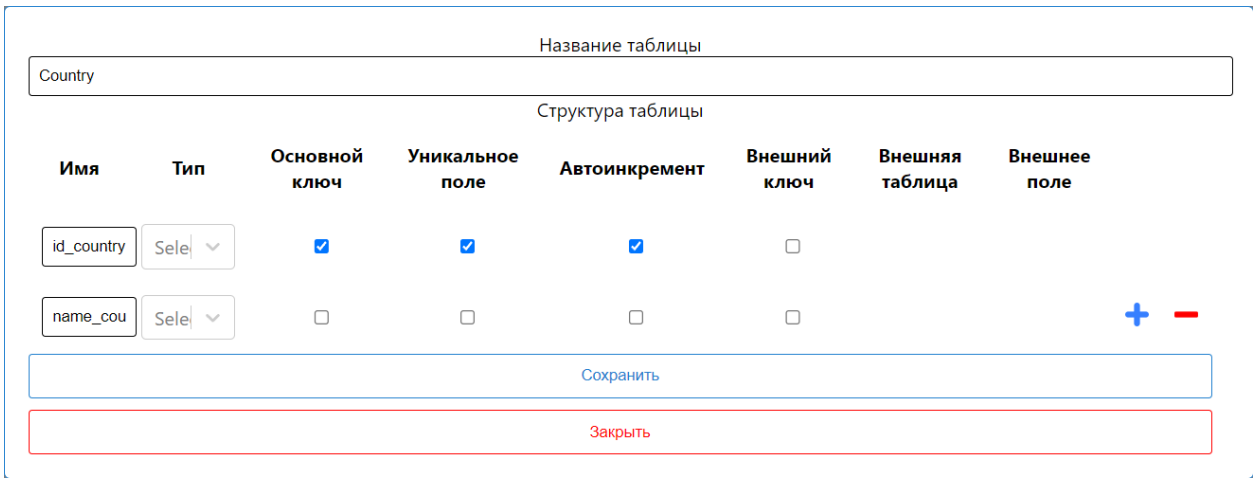


Рисунок 33 – Форма создания таблицы country

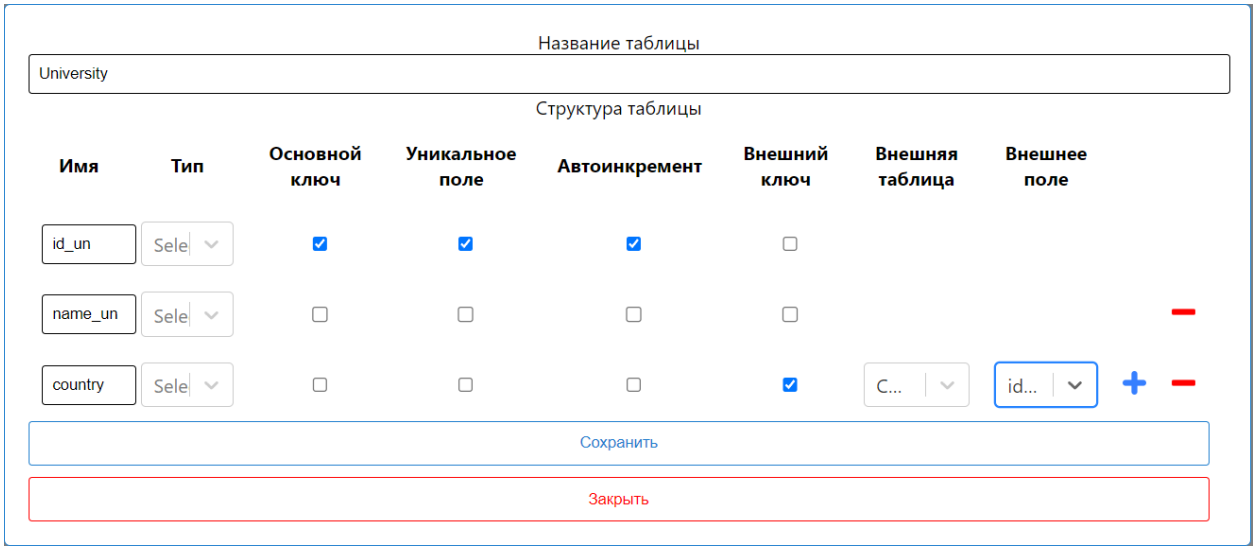


Рисунок 34 – Форма создания таблицы university

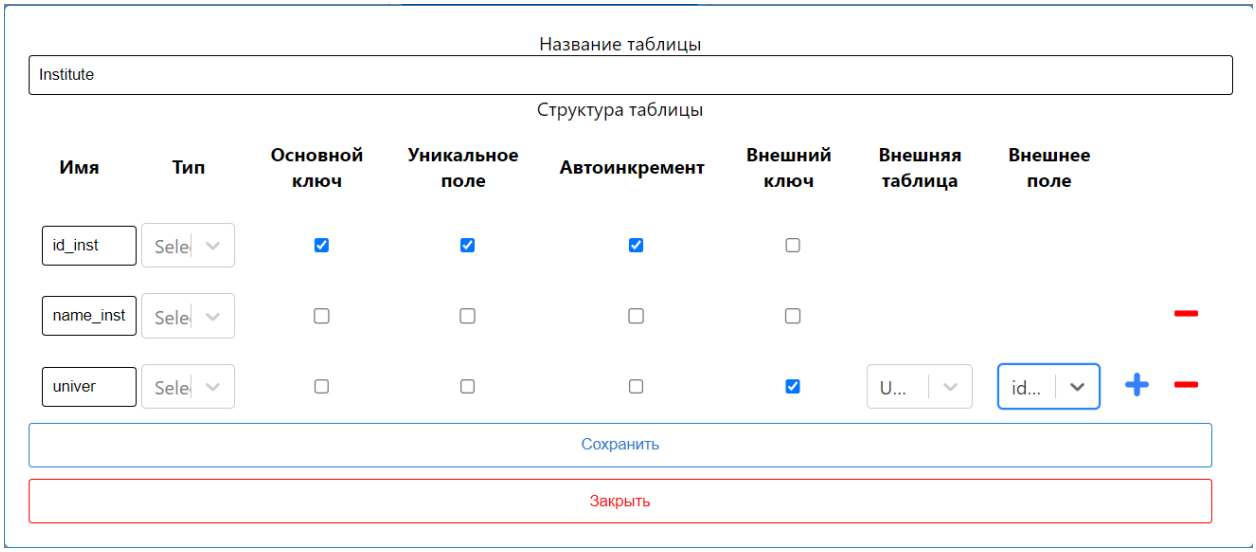


Рисунок 35 – Форма создания таблицы institute

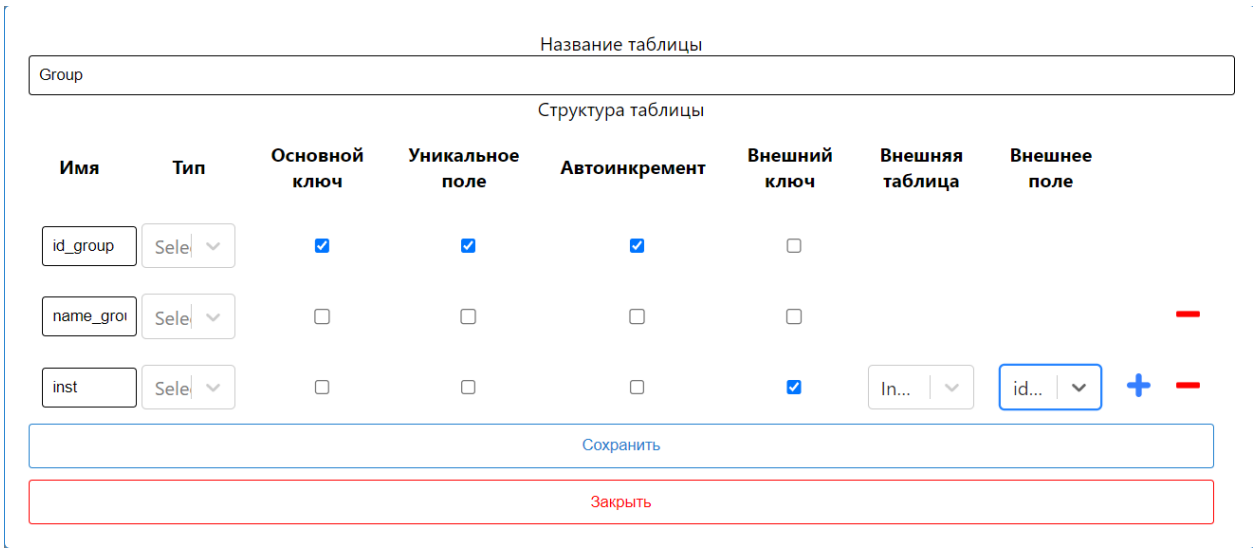


Рисунок 36 – Форма создания таблицы group

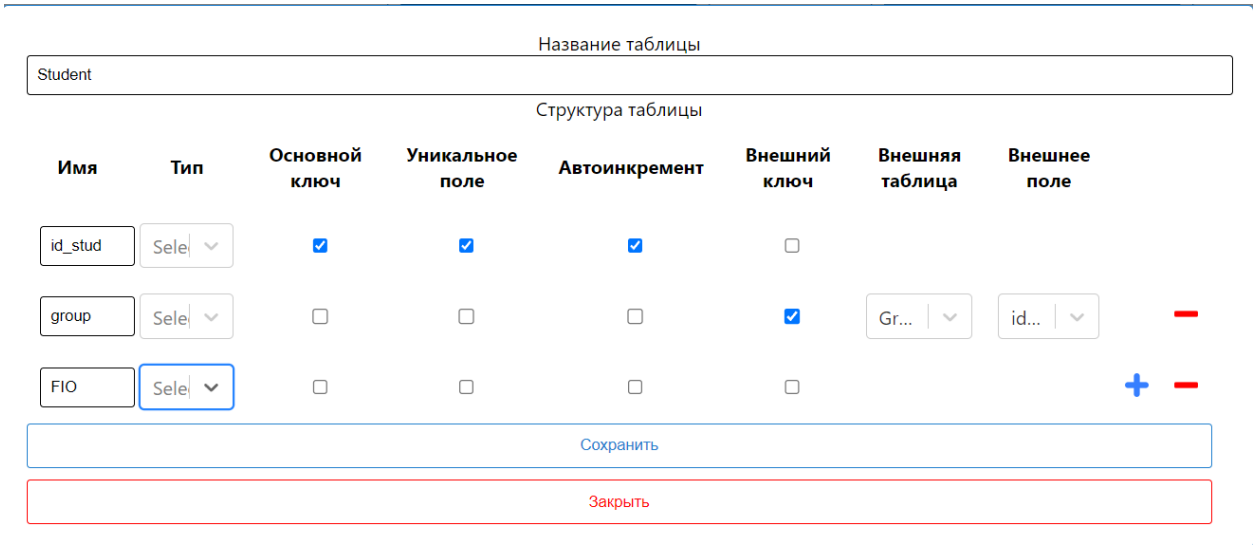


Рисунок 37 – Форма создания таблицы student

```sql
CREATE TABLE Country (
    id_country integer PRIMARY KEY UNIQUE,
    name_country string
);

CREATE TABLE University (
    id_un integer PRIMARY KEY UNIQUE,
    name_un string,
    country integer, FOREIGN KEY (country) REFERENCES Country(id_country)
);

CREATE TABLE Institute (
    id_inst integer PRIMARY KEY UNIQUE,
    name_inst string,
    univer integer, FOREIGN KEY (univer) REFERENCES University(id_un)
);

CREATE TABLE Group (
    id_group integer PRIMARY KEY UNIQUE,
    name_group string,
    inst integer, FOREIGN KEY (inst) REFERENCES Institute(id_inst)
);

CREATE TABLE Student (
    id_stud integer PRIMARY KEY UNIQUE,
    group integer, FOREIGN KEY (group) REFERENCES Group(id_group),
    FIO string
);
```

Рисунок 38 – Тест 1

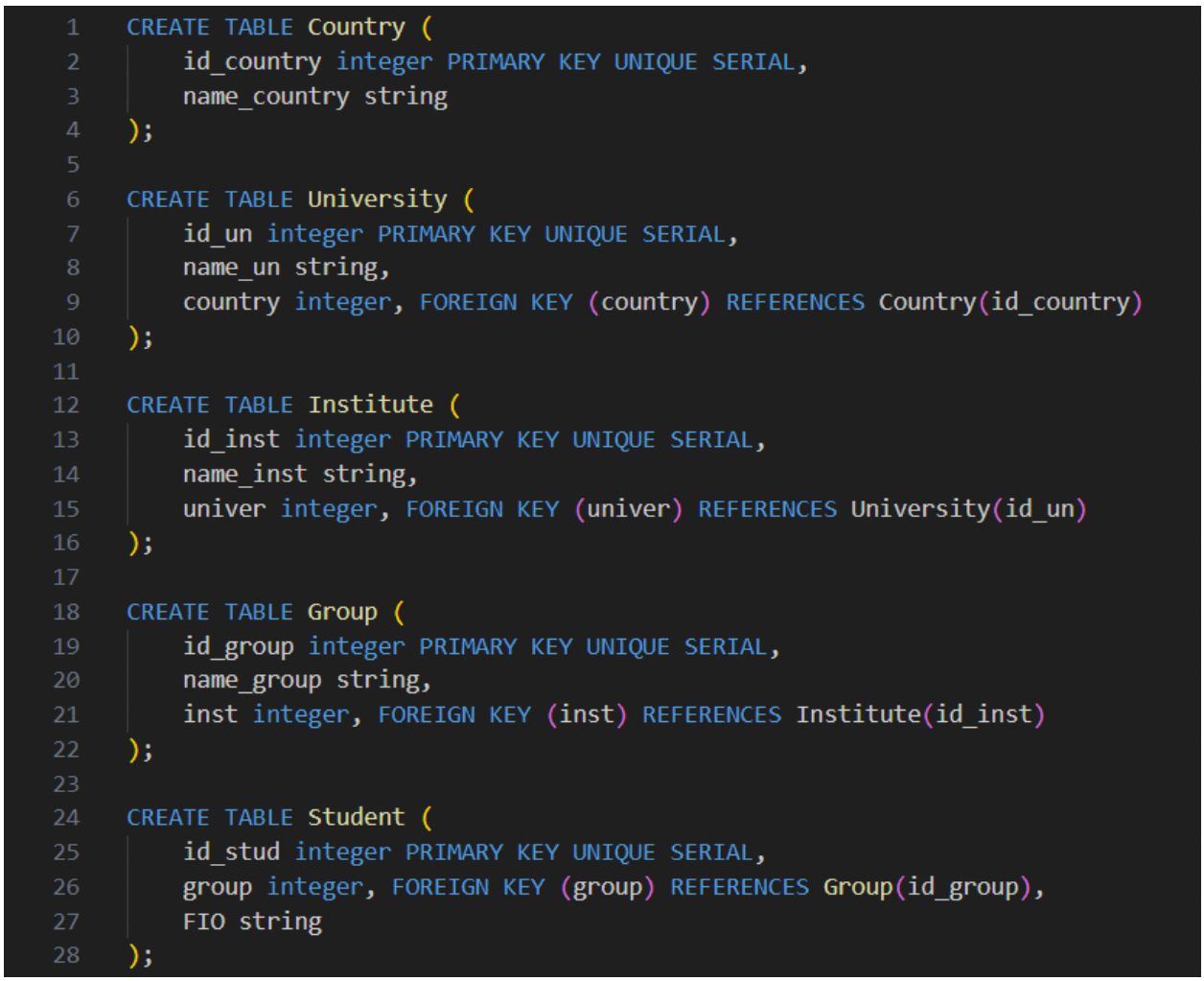
```sql
CREATE TABLE Country (
    id_country integer PRIMARY KEY UNIQUE,
    name_country string
);

CREATE TABLE University (
    id_un integer PRIMARY KEY UNIQUE,
    name_un string,
    country integer, FOREIGN KEY (country) REFERENCES Country(id_country)
);

CREATE TABLE Institute (
    id_inst integer PRIMARY KEY UNIQUE,
    name_inst string,
    univer integer, FOREIGN KEY (univer) REFERENCES University(id_un)
);

CREATE TABLE Group (
    id_group integer PRIMARY KEY UNIQUE,
    name_group string,
    inst integer, FOREIGN KEY (inst) REFERENCES Institute(id_inst)
);

CREATE TABLE Student (
    id_stud integer PRIMARY KEY UNIQUE,
    group integer, FOREIGN KEY (group) REFERENCES Group(id_group),
    FIO string
);
```

Рисунок 39 – Тест 2

```
CREATE TABLE Country (
    id_country integer PRIMARY KEY UNIQUE IDENTITY(1,1),
    name_country string
);

CREATE TABLE University (
    id_un integer PRIMARY KEY UNIQUE IDENTITY(1,1),
    name_un string,
    country integer, FOREIGN KEY (country) REFERENCES Country(id_country)
);

CREATE TABLE Institute (
    id_inst integer PRIMARY KEY UNIQUE IDENTITY(1,1),
    name_inst string,
    univer integer, FOREIGN KEY (univer) REFERENCES University(id_un)
);

CREATE TABLE Group (
    id_group integer PRIMARY KEY UNIQUE IDENTITY(1,1),
    name_group string,
    inst integer, FOREIGN KEY (inst) REFERENCES Institute(id_inst)
);

CREATE TABLE Student (
    id_stud integer PRIMARY KEY UNIQUE IDENTITY(1,1),
    group integer, FOREIGN KEY (group) REFERENCES Group(id_group),
    FIO string
);
```

```
CREATE TABLE color (
    id_color integer PRIMARY KEY UNIQUE IDENTITY(1,1),
    name_color string
);

CREATE TABLE house (
    id_house integer PRIMARY KEY UNIQUE IDENTITY(1,1),
    id_color integer, FOREIGN KEY (id_color) REFERENCES color(id_color)
);
```

Рисунок 40 – Тест 3



```
CREATE TABLE Country (
    id_country integer PRIMARY KEY UNIQUE SERIAL,
    name_country string
);

CREATE TABLE University (
    id_un integer PRIMARY KEY UNIQUE SERIAL,
    name_un string,
    country integer, FOREIGN KEY (country) REFERENCES Country(id_country)
);

CREATE TABLE Institute (
    id_inst integer PRIMARY KEY UNIQUE SERIAL,
    name_inst string,
    univer integer, FOREIGN KEY (univer) REFERENCES University(id_un)
);

CREATE TABLE Group (
    id_group integer PRIMARY KEY UNIQUE SERIAL,
    name_group string,
    inst integer, FOREIGN KEY (inst) REFERENCES Institute(id_inst)
);

CREATE TABLE Student (
    id_stud integer PRIMARY KEY UNIQUE SERIAL,
    group integer, FOREIGN KEY (group) REFERENCES Group(id_group),
    FIO string
);
```

Рисунок 41 – Тест 4

Как мы видим на всех 4 диалектах SQL всё работает корректно.

Заключение

В ходе данной работы был представлен проект, направленный на разработку приложения с фокусом на упрощении процесса создания SQL скриптов для работы с базами данных. Основной целью проекта было создание интуитивно понятного пользовательского интерфейса, который позволил бы разработчикам визуально создавать, модифицировать и управлять структурой баз данных, минуя необходимость ручного написания SQL кода.

Процесс разработки охватывал несколько ключевых этапов, начиная с анализа требований и заканчивая оценкой качества. В рамках анализа требований детально изучили потребности пользователей и основные задачи, которые приложение должно было решать. Это позволило определить основные функциональные возможности и параметры конфигурации, которые должны были быть включены в приложение.

Проектирование архитектуры играло ключевую роль в создании приложения. Разработали общую архитектуру, учитывая особенности работы с базами данных и требования к пользовательскому интерфейсу. Важным аспектом проектирования была гибкость и расширяемость приложения, чтобы обеспечить возможность дальнейшего развития и добавления новых функциональных возможностей.

В ходе реализации функциональности уделяли особое внимание разработке удобного и интуитивно понятного пользовательского интерфейса. Также создали мощный движок для работы с базами данных, который позволял выполнять различные операции без необходимости в написании SQL кода вручную.

Тестирование и оценка качества играли важную роль в обеспечении надежности и производительности приложения. Провели обширное тестирование для обнаружения и исправления возможных ошибок, а также оценили общее качество приложения с учетом его производительности и удобства использования.

Проект направлен на предоставление разработчикам мощного инструмента для работы с базами данных, способствующего повышению их производительности и качества работы. Стремились создать приложение, которое было бы не только эффективным инструментом для создания и управления базами данных, но и интуитивно понятным и легким в использовании.

Список использованных источников

1. Коннолли Т., Бегг К. Базы данных: проектирование, реализация и сопровождение. 2009.
2. Дорофеев А.С. Базы данных: учебное пособие. ИрГТУ, 2008. 80 с.
3. Лесников М. React.js: Приложения для интернета вещей.
4. Дюккер Э. Node.js для профессионалов.
5. Фармаковский Д. PostgreSQL. Эффективное использование.
6. Макбрайд С.Б., Шевченко А.М. SQL. Полное руководство.
7. Степин А. Основы SQL. Практическое руководство по манипулированию данными. 2019.
8. Фейерштейн С. Oracle PL/SQL. Справочник разработчика. 2014.
9. Дольников Д. PostgreSQL. Администрирование и разработка. 2019.
10. Бек А. SQL для чайников. Полное руководство. 2016.
11. Маллен К.С. Базы данных: от реляционных к NoSQL. Введение в базы данных. 2013.
12. Бобруйко В. Учебник по Microsoft SQL Server 2016. 2017.
13. PostgreSQL Documentation. [Электронный ресурс]. URL: https://www.postgresql.org/docs/ (дата обращения 17.05.2024).
14. React Documentation. [Электронный ресурс]. URL: https://reactjs.org/docs/getting-started.html (дата обращения 17.05.2024).
15. Node.js Documentation. [Электронный ресурс]. URL: https://nodejs.org/en/docs/ (дата обращения 17.05.2024).
16. MySQL Workbench [Электронный ресурс]. URL: https://www.mysql.com/ (дата обращения 17.05.2024).
17. Microsoft Visio [Электронный ресурс]. URL: https://www.microsoft.com/ru-ru/microsoft-365/visio/flowchart-software (дата обращения 17.05.2024).
18. Lucidchart [Электронный ресурс]. URL: https://www.lucidchart.com/pages/ru (дата обращения 17.05.2024).
19. Draw.io [Электронный ресурс]. URL: https://app.diagrams.net/ (дата обращения 17.05.2024).
20. Dbdiagram [Электронный ресурс]. URL: https://dbdiagram.io/home (дата обращения 17.05.2024).

Приложение А Программный код

App.js

```
import './App.css';
import Header from './components/Header/Header';
import Footer from './components/Footer/Footer';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import React, { useState } from 'react';
import Home from './pages/Home/Home';


function App() {
  const [tableData, setTableData] = useState([]);

  return (
    <Router>
      <div className="App">
        <Header setTableData={setTableData} tableData={tableData} />
        <Routes>
          <Route path="/" element={<Home tableData={tableData}
setTableData={setTableData} />} />
        </Routes>
        <Footer />
      </div>
    </Router>
  );
}

export default App;
```

Home.jsx

```
import React from 'react';
import MainTable from '../../components/MainTable/MainTable';

const Home = ({ tableData, setTableData }) => {

    console.log('Данные home:');
    console.log(tableData);
    return (
        <div className="Home">
            <MainTable tableData={tableData} setTableData={setTableData}/>
        </div>
    );
};
export default Home;
```

Header.jsx

```
import bar from './bar.png';
import tb from './tb.png';
import note from './note.png';
import im from './im.png';
import ex from './ex.png';
import back from './back.png';
import forward from './forward.png';
import './Header.css';
import Menu from '../Menu/Menu';
import Sticker from '../Sticker/Sticker';
import React, { useState } from 'react';
import Create from '../Create/Create';
import Export from '../Export/Export';
import Import from '../Import/Import';
import Register from '../Register/Register';
import Login from '../Login/Login';
import Table from '../Table/Table';


const Header = ({ setTableData, tableData }) => {
    const [isGuestMenuOpen, setGuestMenuOpen] = useState(false);
    const [isBarMenuOpen, setBarMenuOpen] = useState(false);
    const [stickers, setStickers] = useState([]);
    const [isCreateOpen, setIsCreateOpen] = useState(false);
    const [isExportOpen, setExportOpen] = useState(false);
    const [isImportOpen, setImportOpen] = useState(false);
    const [isRegOpen, setRegOpen] = useState(false);
    const [isLoginOpen, setLoginOpen] = useState(false);
    const [isTableOpen, setTableOpen] = useState(false);

    const handleTableData = (data) => {
        setTableData(data);
      };

    const toggleTablePopup = () => {
        setTableOpen(true);
    }

    const toggleCreatePopup = () => {
        setIsCreateOpen(true);
    };

    const toggleExportPopup = () => {
        setExportOpen(true);
    };

    const toggleImportPopup = () => {
        setImportOpen(true);
```

```
    };

    const toggleRegPopup = () => {
        setRegOpen(true);
        setLoginOpen(false);
    };

    const toggleLoginPopup = () => {
        setLoginOpen(true);
        setRegOpen(false);
    };

    const toggleGuestMenu = () => {
        setGuestMenuOpen(!isGuestMenuOpen);
    };

    const toggleBarMenu = () => {
        setBarMenuOpen(!isBarMenuOpen);
    };

    const handleIconClick = () => {
        const newStickers = [...stickers, <Sticker key={stickers.length}
onDelete={() => handleStickerDelete(stickers.length)} />];
        setStickers(newStickers);
    };

    const handleStickerDelete = (index) => {
        const updatedStickers = stickers.filter((sticker, i) => i !== index);
        setStickers(updatedStickers);
    };

    return (
        <header className='Header'>
            {isCreateOpen && <Create />}
            {isExportOpen && <Export tableData={tableData} />}
            {isImportOpen && <Import />}
            {isRegOpen && <Register toggleLoginPopup={toggleLoginPopup} />}
            {isLoginOpen && <Login toggleRegPopup={toggleRegPopup} />}
            {isTableOpen && <Table setTableOpen={setTableOpen}
setTableData={setTableData} tableData={tableData}/>}
            <div className='Bar' onClick={toggleBarMenu}>
                <img src={bar} alt='Бар' />
                {isBarMenuOpen && <Menu toggleCreatePopup={toggleCreatePopup} />}
            </div>
            <div className='Steps'>
                <div className='Back'>
                    <img src={back} alt='Назад' />
                </div>
                <div className='Forward'>
                    <img src={forward} alt='Вперёд' />
```

```
                </div>
            </div>
            <div className='NT'>
                <div className="Note">
                    <img src={note} alt='Стикер' onClick={handleIconClick} />
                    {stickers.map((sticker, index) => (
                        <React.Fragment key={index}>{sticker}</React.Fragment>
                    ))}
                </div>
                <div className='Table' onClick={toggleTablePopup}>
                    <img src={tb} alt='Таблица' />
                </div>
            </div>
            <div className='im-ex'>
                <div className='Import' onClick={toggleImportPopup}>
                    <img src={im} alt='Импорт' />
                </div>
                <div className='Export' onClick={toggleExportPopup}>
                    <img src={ex} alt='Экспорт' />
                </div>
            </div>
            <div className='Name'>
                DBplanner
            </div>
            <div className='Login-link' onClick={toggleGuestMenu}>
                {isGuestMenuOpen ? (
                    <>
                        <a onClick={toggleLoginPopup}>Войти</a>
                        <a onClick={toggleRegPopup}>Зарегистрироваться</a>
                    </>
                ) : (
                    <span>Гость</span>
                )}
            </div>
        </header>
    );
};
export default Header;
```

Header.css

```
.Header {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  height: 60px;
  background-color: white;
  display: flex;
  justify-content: space-between;
```

```
  align-items: center;
  padding: 10px 10px;
  border-bottom: 5px solid #2180CF;
  z-index: 3;
}

.Bar{
  display: flex;
  align-items: center;
}

.Login-link {
  font-size: 35px;
  padding: 10px 25px;
  margin-left: auto;
  cursor: pointer;
  position: relative;
  color: #2180CF;
}

.Login-link a {
  color: #2180CF;
  text-decoration: none;
  display: block;
  padding: 8px 12px;
  transition: background-color 0.3s ease;
  font-size: 20px;
}

.Login-link a:hover {
  color: #ffffff;
  background-color: #2180CF;
}

.Login-link span::after {
  content: '\25BC';
  font-size: 12px;
}

.Login-link.collapsed span::after {
  content: '\25BC';
}

.Login-link ul {
  list-style: none;
  padding: 0;
  margin: 0;
  position: absolute;
  top: 100%;
  left: 0;
```

```
  right: 0;
  display: none;
  font-size: 12px;
}

.Login-link.open ul {
  display: block;
}

.Bar img {
  cursor: pointer;
  width: 50px;
}

.Table img {
  cursor: pointer;
  width: 50px;
}

.Note img {
  cursor: pointer;
  width: 50px;
}

.Import img {
  cursor: pointer;
  width: 50px;
}

.Export img {
  cursor: pointer;
  width: 50px;
}

.Back img {
  cursor: pointer;
  width: 50px;
}

.Forward img {
  cursor: pointer;
  width: 50px;
}

.NT {
  display: flex;
  justify-content: space-between;
  padding: 5px 5px;
  border-right: 5px solid #2180CF;
  border-left: 5px solid #2180CF;
```

```
}

.Note,
.Table {
  flex: 5;
  position: relative;
}

.Export,
.Import {
  flex: 5;
  padding: 5px;
}

.im-ex {
  display: flex;
  justify-content: space-between;
  padding: 5px 5px;
}

.Name {
  display: flex;
  color: #2180CF;
  font-size: 35px;
  text-align: center;
  margin: 0 auto;
}

.Back,
.Forward {
  flex: 5;
  padding: 5px;
}

.Steps {
  display: flex;
  align-items: center;
  padding: 5px 5px;
}
```

Register.jsx

```
import './Register.css';
import React, { useState } from 'react';
import axios from 'axios';

const Register = ({ toggleLoginPopup }) => {
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');

    const handleRegister = async () => {
        try {
            const response = await axios.post('/register', { email, password });
            console.log(response.data.message);
        } catch (error) {
            console.error('Ошибка при регистрации:',
error.response.data.message);
        }
    };

    const handleToggleLoginPopup = () => {
        toggleLoginPopup();
    };

    return (
        <div className='Register'>
            <form onSubmit={handleRegister}>
                <div>
                    <label>Email</label>
                    <input type="email" value={email} onChange={(e) =>
setEmail(e.target.value)} />
                </div>
                <div>
                    <label>Пароль</label>
                    <input type="password" value={password} onChange={(e) =>
setPassword(e.target.value)} />
                </div>
                <div className='RegBtn'>
                    <button>Зарегистрироваться</button>
                </div>
                <div className='ClBtn'>
                    <button>Закрыть</button>
                </div>
                <p>
                    Есть аккаунт?
                    <a onClick={handleToggleLoginPopup}>
                        Войдите
                    </a>
                </p>
            </form>
```

```
        </div>
    );
};

export default Register;
```

Register.css

```
.Register {
    position: fixed;
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    height: 100vh;
    top: 0;
    left: 0;
    right: 0;
    bottom: 0;
    z-index: 999;
    background-color: rgba(0, 0, 0, 0.5)
}

.Register form {
    width: 300px;
    padding-left: 20px;
    padding-right: 20px;
    padding-top: 20px;
    border: 1px solid #2180CF;
    border-radius: 5px;
    background-color: #fff;
}

.Register form div {
    margin-bottom: 5px;
    margin-right: 15px;
}

.Register form label {
    display: block;
    margin-bottom: 10px;
}

.Register form input {
    width: 100%;
    padding: 8px;
    border: 1px solid black;
    border-radius: 3px;
}
```

```css
.RegBtn button {
    width: 105%;
    padding: 10px;
    background-color: #fff;
    color: #2180CF;
    border: none;
    border-radius: 3px;
    cursor: pointer;
    margin-top: 10px;
    border: 1px solid #2180CF;
    transition-duration: 0.4s;
}

.RegBtn button:hover {
    background-color: #2180CF;
    color: #fff;
    border-radius: 3px;
    cursor: pointer;
}

.ClBtn button {
    width: 105%;
    padding: 10px;
    background-color: #fff;
    color: red;
    border: none;
    border-radius: 3px;
    cursor: pointer;
    margin-top: 10px;
    border: 1px solid red;
    transition-duration: 0.4s;
}

.ClBtn button:hover {
    background-color: red;
    color: #fff;
    border-radius: 3px;
    cursor: pointer;
}

.Register p a {
    margin-left: 5px;
    color: #2180CF;
    cursor: pointer;
}
```

Login.jsx

```
import './Login.css';
import React, { useState } from 'react';
import axios from 'axios';

const Login = ({ toggleRegPopup }) => {
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');

    const handleLogin = async () => {
        try {
            const response = await axios.post('/login', { email, password });
            console.log(response.data.message);
        } catch (error) {
            console.error('Ошибка при входе:', error.response.data.message);
        }
    };

    const handleToggleRegPopup = () => {
        toggleRegPopup();
    };

    return (
        <div className='Login'>
            <form onSubmit={handleLogin}>
                <div>
                    <label>Email</label>
                    <input type="email" value={email} onChange={(e) =>
setEmail(e.target.value)} />
                </div>
                <div>
                    <label>Пароль</label>
                    <input type="password" value={password} onChange={(e) =>
setPassword(e.target.value)} />
                </div>
                <div className='RegBtn'>
                    <button>Войти</button>
                </div>
                <div className='ClBtn'>
                    <button>Закрыть</button>
                </div>
                <p>
                    Нету аккаунта?
                    <a onClick={handleToggleRegPopup}>
                        Зарегистрируйтесь
                    </a>
                </p>
            </form>
        </div>
```

```
    );
};

export default Login;
```

Login.css

```
.Login {
  position: fixed;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  height: 100vh;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  z-index: 999;
  background-color: rgba(0, 0, 0, 0.5)
}

.Login form {
  width: 300px;
  padding-left: 20px;
  padding-right: 20px;
  padding-top: 20px;
  border: 1px solid black;
  border-radius: 5px;
  background-color: #fff;
}

.Login form div {
  margin-bottom: 5px;
  margin-right: 15px;
}

.Login form label {
  display: block;
  margin-bottom: 10px;
}

.Login form input {
  width: 100%;
  padding: 8px;
  border: 1px solid #2180CF;
  border-radius: 3px;
}

.Login p a {
```

```
  margin-left: 5px;
  color: #2180CF;
  cursor: pointer;
}
```

Table.jsx

```
import React, { useState } from 'react';
import './Table.css';
import Select from 'react-select';
import plus from './blue-plus.png';
import minus from './red-minus.png';

const Table = ({ tableData, setTableData, setTableOpen }) => {
  const [tableName, setTableName] = useState('');
  const [tableRows, setTableRows] = useState([
    {
      name: '',
      type: null,
      isPrimaryKey: false,
      isUnique: false,
      isAutoIncrement: false,
      isForeignKey: false,
      foreignTable: null,
      foreignField: null,
    },
  ]);

  const options = [
    { value: 'integer', label: 'integer' },
    { value: 'string', label: 'string' },
    { value: 'char', label: 'char' },
  ];


  const handleInputChange = (index, field, value) => {
    const updatedRows = [...tableRows];
    updatedRows[index][field] = value;
    if (field === 'isForeignKey') {
      if (value) {
        updatedRows[index].foreignTable = null;
        updatedRows[index].foreignField = null;
      }
    }
    setTableRows(updatedRows);
  };

  const addRow = () => {
    setTableRows([
```

```
      ...tableRows,
      {
        name: '',
        type: null,
        isPrimaryKey: false,
        isUnique: false,
        isAutoIncrement: false,
        isForeignKey: false,
        foreignTable: null,
        foreignField: null,
      },
    ]);
  };

  const deleteRow = (index) => {
    const updatedRows = [...tableRows];
    updatedRows.splice(index, 1);
    setTableRows(updatedRows);
  };

  const handleSubmit = (e) => {
    if (!tableName.trim()) {
      alert("Введите название таблицы.");
      return;
    }

    if (
      tableRows[0].name.trim() === '' ||
      tableRows[0].type === null
    ) {
      alert("Заполните первые 2 поля первой строки.");
      return;
    }
    e.preventDefault();
    const tableIdCounter = Date.now();
    const newTableData = { id: tableIdCounter, name: tableName, rows: tableRows
};
    setTableData(prevTableData => [...prevTableData, newTableData]);
    console.log('Отправленные данные:');
    console.log(newTableData);
    setTableName('');
    setTableRows([{ /* reset rows to initial state */ }]);
    setTableOpen(false);
  };


  return (
    <div className='TableCreator'>
      <form onSubmit={handleSubmit}>
        <label>Название таблицы</label>
```

```
        <div>
          <input type="text" value={tableName} onChange={(e) =>
setTableName(e.target.value)} />
        </div>
        <label>Структура таблицы</label>
        <table>
          <thead>
            <tr>
              <th>Имя</th>
              <th>Тип</th>
              <th>Основной ключ</th>
              <th>Уникальное поле</th>
              <th>Автоинкремент</th>
              <th>Внешний ключ</th>
              <th>Внешняя таблица</th>
              <th>Внешнее поле</th>
              <th></th>
            </tr>
          </thead>
          <tbody>
            {tableRows.map((row, index) => (
              <tr key={index}>
                <td><input type="text" value={row.name} onChange={(e) =>
handleInputChange(index, 'name', e.target.value)} /></td>
                <td><Select options={options} value={row.type}
onChange={(selectedOption) => handleInputChange(index, 'type',
selectedOption.value)} /></td>
                <td><input type="checkbox" checked={row.isPrimaryKey}
onChange={(e) => handleInputChange(index, 'isPrimaryKey', e.target.checked)}
/></td>
                <td><input type="checkbox" checked={row.isUnique} onChange={(e)
=> handleInputChange(index, 'isUnique', e.target.checked)} /></td>
                <td><input type="checkbox" checked={row.isAutoIncrement}
onChange={(e) => handleInputChange(index, 'isAutoIncrement', e.target.checked)}
/></td>
                <td><input type="checkbox" checked={row.isForeignKeyforeignKey}
onChange={(e) => handleInputChange(index, 'isForeignKey', e.target.checked)}
/></td>
                <td>
                  {row.isForeignKey && tableData && (
                    <Select
                      options={tableData.map((table) => ({ value: table.id,
label: table.name }))}
                      value={row.foreignTable}
                      onChange={(selectedOption) => handleInputChange(index,
'foreignTable', selectedOption)}
                    />
                  )}
                </td>
                <td>
```

```
                {row.isForeignKey && (
                  <Select
                    options={row.foreignTable ? tableData.find(table =>
table.id === row.foreignTable.value).rows.map(row => ({ value: row.name, label:
row.name })) : []}
                    value={row.foreignField}
                    onChange={(selectedOption) => handleInputChange(index,
'foreignField', selectedOption)}
                  />
                )}
              </td>
              <td>{index === tableRows.length - 1 ? <img src={plus} alt='Плюс'
style={{ width: '20px', height: '20px', cursor: 'pointer' }} onClick={addRow} />
: null}</td>
              <td>
                {index === 0 ? null : (
                  <img src={minus} alt='Минус' style={{ width: '20px', height:
'20px', cursor: 'pointer' }} onClick={deleteRow} />
                )}
              </td>
            </tr>
          ))}
        </tbody>
      </table>
      <div className='Buttons'>
        <div className='CreateBtn'>
          <button type="submit">Сохранить</button>
        </div>
        <div className='CloseBtn'>
          <button type="button">Закрыть</button>
        </div>
      </div>
    </form>
  </div>
 );
};

export default Table;
```

Table.css

```
.TableCreator {
  position: fixed;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  height: 100vh;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  z-index: 999;
  background-color: rgba(0, 0, 0, 0.5)
}

.TableCreator form {
  width: 1000px;
  padding: 20px;
  padding-right: 35px;
  border: 1px solid #2180CF;
  border-radius: 5px;
  background-color: white;
}

.TableCreator form input {
  width: 100%;
  padding: 8px;
  border: 1px solid black;
  border-radius: 3px;
}

.CreateBtn button{
  width: 100%;
  padding: 10px;
  background-color: white;
  color: #2180CF;
  border: none;
  border-radius: 3px;
  cursor: pointer;
  border: 1px solid #2180CF;
  transition-duration: 0.4s;
}

.CreateBtn button:hover {
  background-color: #2180CF;
  color: #fff;
}
```

```css
.CloseBtn button{
  width: 100%;
  padding: 10px;
  background-color: #fff;
  color: red;
  border: none;
  border-radius: 3px;
  cursor: pointer;
  margin-top: 10px;
  border: 1px solid red;
  transition-duration: 0.4s;
}

.CloseBtn button:hover {
  background-color: red;
  color: #fff;
}

.DeleteBtn button{
  width: 100%;
  padding: 10px;
  background-color: red;
  color: white;
  border: none;
  border-radius: 3px;
  cursor: pointer;
  margin-top: 10px;
  border: 1px solid red;
  transition-duration: 0.4s;
}

.DeleteBtn button:hover {
  background-color: white;
  color: red;
}

.TableCreator form th {
  padding: 15px;
}

.TableCreator form td {
  padding: 10px;
}
```

MainTable.jsx

```
import React, { useState } from 'react';
import './MainTable.css';
import EditTable from '../EditTable/EditTable';
import Draggable from 'react-draggable';

const MainTable = ({ tableData, setTableData }) => {

  const [isEditTableOpen, setIsEditTableOpen] = useState(false);
  const [currentTableName, setCurrentTableName] = useState('');
  const [currentTableRows, setCurrentTableRows] = useState([]);
  const [selectedTableId, setSelectedTableId] = useState(null);

  const handleTableDoubleClick  = (tableIdCounter, tableName, tableRows) => {
    setSelectedTableId(tableIdCounter);
    setCurrentTableName(tableName);
    setCurrentTableRows(tableRows);
    setIsEditTableOpen(true);
  };

  const handleDeleteTable = () => {
    const updatedTableData = tableData.filter(table => table.id !==
selectedTableId);
    setTableData(updatedTableData);
    setIsEditTableOpen(false);
  };

  console.log('Пришедшие данные:');
  console.log(tableData);

  if (!tableData || !tableData.length) {
    return null;
  }

  return (
    <div className="MainTable">
      {isEditTableOpen && (
        <EditTable
          tableName={currentTableName}
          tableRows={currentTableRows}
          onClose={() => setIsEditTableOpen(false)}
          onDelete={handleDeleteTable}
        />
      )}
      {tableData.map((table, index) => (
        <Draggable key={index}>
          <form key={table.id} onDoubleClick={() => handleTableDoubleClick
(table.id, table.name, table.rows)}>
            <h2>{table.name}</h2>
```

```
            <table>
              <thead>
                <tr>
                  <th>Имя</th>
                  <th>Тип</th>
                </tr>
              </thead>
              <tbody>
                {table.rows.map((row, index) => (
                  <tr key={index}>
                    <td>{row.name}</td>
                    <td>{row.type}</td>
                  </tr>
                ))}
              </tbody>
            </table>
          </form>
        </Draggable>
      ))}
    </div>
  );
};

export default MainTable;
```

MainTable.css

```
.MainTable {
  position: fixed;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
}


.MainTable form {
  width: 250px;
  padding: 10px;
  border: 1px solid #2180CF;
  border-radius: 5px;
  background-color: white;
}


.MainTable form th {
```

```
  padding: 5px;
}

.MainTable form td {
  padding: 5px;
}

h2 {
  background-color: #2180CF;
  color: white;
}
```

Export.jsx

```
import './Export.css';
import React, { useState, useEffect } from 'react';
import Select from 'react-select'
import axios from 'axios';


const Export = ({ tableData }) => {

    const [sqlOptions, setSQLOptions] = useState([]);

    useEffect(() => {
        const fetchSQLOptions = async () => {
            try {
                const response = await
axios.get(`http://localhost:4000/sql_type`);
                const data = response.data.map(item => ({
                    value: item.name_sql,
                    label: item.name_sql,
                  }));
                setSQLOptions(data);
            } catch (error) {
                console.error('Ошибка при загрузке данных:', error);
            }
        };

        fetchSQLOptions();
    }, []);

    console.log('Данные экспорт:',tableData);

    const [selectedSQLType, setSelectedSQLType] = useState(null);

    const handleGenerateSQL = () => {
        if (selectedSQLType) {
            const generateSQLScript = (tableData, selectedSQLType) => {
```

```
                let sqlScript = '';

                if (selectedSQLType === 'PostgreSQL') {

                    for (let i = 0; i < tableData.length; i++) {
                        let table = tableData[i];
                        let tableScript = `CREATE TABLE ${table.name} (\n`;
                        for (let j = 0; j < table.rows.length; j++) {
                            let row = table.rows[j];
                            let columnScript = `${row.name} ${row.type}`;
                            if (row.isPrimaryKey) {
                                columnScript += ' PRIMARY KEY';
                            }
                            if (row.isUnique) {
                                columnScript += ' UNIQUE';
                            }
                            if (row.isAutoIncrement) {
                                columnScript += ' SERIAL';
                            }
                            if (row.isForeignKey && row.foreignTable &&
row.foreignField) {

                                columnScript += `, FOREIGN KEY (${row.name})
REFERENCES ${row.foreignTable.name}(${row.foreignField.name})`;
                              }
                            if (j !== table.rows.length - 1) {
                                columnScript += ',';
                            }
                            tableScript += `\t${columnScript}\n`;
                        }
                        tableScript += `);\n\n`;
                        sqlScript += tableScript;
                    }

                    if (selectedSQLType === 'MySQL') {
                        for (let i = 0; i < tableData.length; i++) {
                            let table = tableData[i];
                            let tableScript = `CREATE TABLE ${table.name} (\n`;
                            for (let j = 0; j < table.rows.length; j++) {
                              let row = table.rows[j];
                              let columnScript = `${row.name} ${row.type}`;
                              if (row.isPrimaryKey) {
                                columnScript += ' PRIMARY KEY';
                              }
                              if (row.isUnique) {
                                columnScript += ' UNIQUE';
                              }
                              if (row.isAutoIncrement) {
                                columnScript += ' AUTO_INCREMENT';
```

```
                            }
                            if (row.isForeignKey && row.foreignTable &&
row.foreignField) {
                              columnScript += `, FOREIGN KEY (${row.name})
REFERENCES ${row.foreignTable.name}(${row.foreignField.name})`;
                            }
                            if (j !== table.rows.length - 1) {
                              columnScript += ',';
                            }
                            tableScript += `\t${columnScript}\n`;
                          }
                          tableScript += `);\n\n`;
                          sqlScript += tableScript;
                        }

                    }

                    if(selectedSQLType === 'Microsoft SQL Server'){
                        for (let i = 0; i < tableData.length; i++) {
                            let table = tableData[i];
                            let tableScript = `CREATE TABLE ${table.name} (\n`;
                            for (let j = 0; j < table.rows.length; j++) {
                              let row = table.rows[j];
                              let columnScript = `${row.name} ${row.type}`;
                              if (row.isPrimaryKey) {
                                columnScript += ' PRIMARY KEY';
                              }
                              if (row.isUnique) {
                                columnScript += ' UNIQUE';
                              }
                              if (row.isAutoIncrement) {
                                columnScript += ' IDENTITY(1,1)';
                              }
                              if (row.isForeignKey && row.foreignTable &&
row.foreignField) {
                                columnScript += `, FOREIGN KEY (${row.name})
REFERENCES ${row.foreignTable.name}(${row.foreignField.name})`;
                              }
                              if (j !== table.rows.length - 1) {
                                columnScript += ',';
                              }
                              tableScript += `\t${columnScript}\n`;
                            }
                            tableScript += `);\n\n`;
                            sqlScript += tableScript;
                          }
                    }

                    if(selectedSQLType === 'Oracle'){
                        for (let i = 0; i < tableData.length; i++) {
```

```
                            let table = tableData[i];
                            let tableScript = `CREATE TABLE ${table.name} (\n`;
                            for (let j = 0; j < table.rows.length; j++) {
                              let row = table.rows[j];
                              let columnScript = `${row.name} ${row.type}`;
                              if (row.isPrimaryKey) {
                                columnScript += ' PRIMARY KEY';
                              }
                              if (row.isUnique) {
                                columnScript += ' UNIQUE';
                              }
                              if (row.isAutoIncrement) {
                              }
                              if (row.isForeignKey && row.foreignTable &&
row.foreignField) {
                                columnScript += `, FOREIGN KEY (${row.name})
REFERENCES ${row.foreignTable.name}(${row.foreignField.name})`;
                              }
                              if (j !== table.rows.length - 1) {
                                columnScript += ',';
                              }
                              tableScript += `\t${columnScript}\n`;
                            }
                            tableScript += `);\n\n`;
                            sqlScript += tableScript;
                          }
                      }

                  }
                  return sqlScript;
              };

              const sqlScript = generateSQLScript(tableData,
selectedSQLType.value);
              console.log('Скрипт:',sqlScript);
              const blob = new Blob([sqlScript], { type: 'text/plain' });
              const url = URL.createObjectURL(blob);

              const link = document.createElement('a');
              link.href = url;
              link.download = 'script.sql';

              link.click();

              URL.revokeObjectURL(url);

          } else {
              alert('Выберите тип SQL');
          }
      };
```

```
    return (
        <div className="ExP">
            <form>
                <div className='ChooseSql'>
                    Выберите тип SQL
                </div>
                <Select
                    options={sqlOptions}
                    value={selectedSQLType}
                    onChange={setSelectedSQLType}
                />
                <div className='GenBtn'>
                    <button onClick={handleGenerateSQL}>Сгенерировать
SQL</button>
                </div>
                <div className='DelBtn'>
                    <button>Закрыть</button>
                </div>
            </form>
        </div>
    );
};

export default Export;
```

Export.css

```
.ExP {
  position: fixed;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  height: 100vh;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  z-index: 999;
  background-color: rgba(0, 0, 0, 0.5)
}

.ExP form {
  width: 300px;
  padding: 20px;
  border: 1px solid #2180CF;
  border-radius: 5px;
  background-color: white;
}
```

```css
.GenBtn button {
  width: 100%;
  padding: 10px;
  background-color: #fff;
  color: #2180CF;
  border: none;
  border-radius: 3px;
  cursor: pointer;
  margin-top: 10px;
  border: 1px solid #2180CF;
  transition-duration: 0.4s;
}

.GenBtn button:hover {
  background-color: #2180CF;
  color: #fff;
  border-radius: 3px;
  cursor: pointer;
}

.DelBtn button {
  width: 100%;
  padding: 10px;
  background-color: #fff;
  color: red;
  border: none;
  border-radius: 3px;
  cursor: pointer;
  margin-top: 10px;
  border: 1px solid red;
  transition-duration: 0.4s;
}

.DelBtn button:hover {
  background-color: red;
  color: #fff;
  border-radius: 3px;
  cursor: pointer;
}

.ExP form input {
  width: 93%;
  padding: 10px;
  border: 1px solid #2180CF;
  border-radius: 3px;
}

.ChooseSql {
  margin-bottom: 15px;
```

```
  padding: 7px;
  border-bottom: 1px solid #2180CF;
}
```

Index.js

```
import express from 'express';
const app = express();
const port = 4000;
import pg from 'pg';
import cors from 'cors';

const { Pool } = pg;

app.use(cors());

const pool = new Pool({
  user: 'postgres',
  host: 'localhost',
  database: 'Diplom',
  password: '12072002',
  port: 1880,
});

app.post('/login', async (req, res) => {
  const { email, password } = req.body;

  try {
    const user = await pool.query('SELECT * FROM "users" WHERE email = $1 AND
password = $2', [email, password]);

    if (user.rows.length === 0) {
      return res.status(401).json({ message: 'Неверный email или пароль' });
    }


    return res.json({ message: 'Вход выполнен успешно' });
  } catch (error) {
    console.error('Ошибка при выполнении запроса:', error);
    res.status(500).send('Ошибка сервера');
  }
});

app.post('/register', async (req, res) => {
  const { email, password } = req.body;

  try {
    const existingUser = await pool.query('SELECT * FROM users WHERE email = $1',
[email]);
    if (existingUser.rows.length > 0) {
```

```
      return res.status(400).json({ message: 'Пользователь с таким email уже
зарегистрирован' });
    }

    const newUser = await pool.query('INSERT INTO users (email, password) VALUES
($1, $2) RETURNING id', [email, password]);
    const userId = newUser.rows[0].id;

    return res.json({ userId });
  } catch (error) {
    console.error('Ошибка при выполнении запроса:', error);
    res.status(500).send('Ошибка сервера');
  }
});


app.get('/sql_type', (req, res) => {

  pool.query('SELECT name_sql FROM "SQL"', (error, result) => {
    if (error) {
      console.error('Ошибка при выполнении запроса:', error);
      res.status(500).send('Ошибка сервера');
    } else {
      res.json(result.rows);
    }
  });
});

app.get('/', (req, res) => {
  res.send('Привет, мир!');
});

app.listen(port, () => {
  console.log(`Сервер запущен на порту ${port}`);
});
```