|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

|  |  |
| --- | --- |
| ФАКУЛЬТЕТ | «Информатика и системы управления» |

|  |  |
| --- | --- |
| КАФЕДРА | «Теоретическая информатика и компьютерные технологии» |

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К КУРСОВОЙ РАБОТЕ***

***НА ТЕМУ:***

***Мобильное приложение для зарядных станций***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент | ИУ9-62Б |  |  |  | Караник А.А. |
|  | (Группа) |  | (Подпись, дата) |  | (И.О. Фамилия) |

|  |  |  |  |
| --- | --- | --- | --- |
| Руководитель курсовой работы |  |  | Царев А.С. |
|  | (Подпись, дата) |  | (И.О. Фамилия) |

*2024*

# СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 2](#_Toc177000629)

[1. Обзор предметной области 3](#_Toc177000630)

[2. Архитектура системы 5](#_Toc177000631)

[3. Описание базы данных 5](#_Toc177000632)

[3.1. Модель «сущность-связь» 5](#_Toc177000633)

[3.2. Реляционная модель 10](#_Toc177000634)

[4. Описание API взаимодействия приложения с сервером 15](#_Toc177000635)

[4.1. Аутентификация пользователя 15](#_Toc177000636)

[4.2. Регистрация пользователя 16](#_Toc177000637)

[4.3. Получения списка зарядных станций 17](#_Toc177000638)

[4.4. Получение подробной информации о зарядной станции 17](#_Toc177000639)

[4.5. Создание заказа 18](#_Toc177000640)

[4.6. Получение информации по заказу 19](#_Toc177000641)

[4.7. Создание отметки по зарядке 19](#_Toc177000642)

[4.8. Подтверждение учетной записи 20](#_Toc177000643)

[4.9. Получение изображения зарядной станции 20](#_Toc177000644)

[5. Стек технологий 21](#_Toc177000645)

[6. Реализация 22](#_Toc177000646)

[7. Тестирование 35](#_Toc177000647)

[ЗАКЛЮЧЕНИЕ 40](#_Toc177000648)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 41](#_Toc177000649)

# ВВЕДЕНИЕ

В последние годы стремительное развитие электромобилей (ЭМ) стало одной из ключевых тенденций в сфере транспорта и энергетики. Прогнозы аналитиков подтверждают, что доля электромобилей на мировом автомобильном рынке будет неуклонно расти, а значит, спрос на инфраструктуру для их обслуживания будет увеличиваться. Одним из важнейших элементов этой инфраструктуры являются зарядные станции, которые обеспечивают бесперебойное и удобное питание электромобилей. Однако с ростом числа ЭМ возникает проблема их своевременной и удобной подзарядки, особенно в условиях городов и на дальних маршрутах. Эффективное управление зарядными станциями становится необходимым условием для дальнейшего распространения и развития электромобильности.

На фоне этих изменений актуальной становится задача создания мобильных приложений, которые могут оптимизировать процесс использования зарядных станций, повысить их доступность и обеспечить комфорт для владельцев ЭМ. Современные мобильные приложения позволяют пользователям в реальном времени находить ближайшие зарядные станции, отслеживать их доступность, бронировать время зарядки и получать информацию о стоимости услуг. Для реализации этих функций необходима надежная система управления данными, что делает использование баз данных ключевым аспектом разработки.

База данных в данном проекте будет использоваться для хранения и управления информацией о зарядных станциях, пользователях, их бронированиях и других связанных данных. Она обеспечит возможность быстрого доступа к необходимой информации, поддерживая актуальность данных и их синхронизацию в реальном времени. Это не только улучшит пользовательский опыт, но и повысит эффективность использования зарядных станций, способствуя оптимизации энергетических ресурсов и снижению нагрузки на инфраструктуру.

Таким образом, тема разработки мобильного приложения с использованием базы данных для зарядных станций представляет собой актуальную задачу, которая объединяет технологии баз данных, транспорт и энергетику в единый комплекс. Целью данного курсового проекта является разработка функционального прототипа мобильного приложения, которое позволит пользователям эффективно находить и использовать зарядные станции для электромобилей. Приложение должно обеспечить удобный интерфейс, предоставлять актуальную информацию о доступных зарядных станциях и возможности бронирования.

В процессе работы будут рассмотрены ключевые аспекты разработки мобильных приложений, включая проектирование структуры базы данных, особенности проектирования пользовательского интерфейса, интеграцию с программным интерфейсом для получения данных о зарядных станциях, а также аспекты тестирования и внедрения приложения. Реализация этого проекта не только подтвердит теоретические знания, но и продемонстрирует практические навыки в области разработки мобильного ПО и работы с базами данных, что особенно важно в условиях быстро развивающегося рынка электромобилей.

## Обзор предметной области

Разработка мобильного приложения для зарядных станций предполагает глубокое понимание предметной области и сущностей, которые необходимо учитывать при проектировании базы данных и функциональности приложения. Ключевыми элементами предметной области являются зарядные станции, коннекторы, пользователи, отметки по зарядке, изображения зарядных станций, типы зарядок и заказы. Рассмотрение этих элементов позволяет создать целостную картину предметной области и определить, каким образом данные будут структурированы и управляться в рамках приложения.

Важнейшим элементом предметной области является зарядная станция. Каждая станция имеет уникальный идентификатор, местоположение, описание, время работы, данные о коннекторах. Информация о зарядных станциях позволяет пользователям находить ближайшие и наиболее подходящие станции для зарядки своих автомобилей.

Неотъемлемой частью зарядных станций являются коннекторы, которые обеспечивают физическое подключение электромобиля к станции. Различные модели электромобилей требуют использования различных типов коннекторов, таких как Type 1, Type 2, CHAdeMO или GB/T. Важными характеристиками коннекторов являются их тип и мощность зарядки. Информация о коннекторах необходима для того, чтобы пользователь мог выбрать станцию и оценить время, необходимое для полной зарядки.

После каждой сессии зарядки пользователю будет предложено создать отметку по зарядке, фиксирующую информацию о процессе зарядки, включая идентификатор пользователя, станции. Эти данные необходимы для учета использования зарядных станций, а также для формирования отчетов.

Пользователь является не менее важной сущностью системы. Приложение будет ориентировано на владельцев электромобилей. Для каждого пользователя требуется хранение идентификационных данных. Эти данные обеспечивают персонализацию работы приложения и позволяют пользователям быстро находить и бронировать зарядные станции и отслеживать статус зарядок.

Изображения зарядных станций предоставляют пользователям визуальную информацию о станциях, что облегчает поиск нужного объекта и ориентацию на месте. Хранение изображений в базе данных с привязкой к конкретной станции позволяет пользователям заранее ознакомиться с внешним видом станции и ее особенностями, что способствует улучшению пользовательского опыта.

Существенную роль в системе играют типы зарядок, которые определяют параметры коннекторов. В зависимости от типа зарядки пользователь может выбрать станцию, оптимально соответствующую его потребностям.

Наконец, важной сущностью являются заказы, которые фиксируют процесс бронирования и зарядки на станции, предоставляя комплексную информацию, необходимую для управления этими процессами. Каждый заказ содержит данные о пользователе, выбранной станции и коннекторе, а также информацию о состоянии зарядки.

## Архитектура системы

Система, разработанная для работы с зарядными станциями, состоит из нескольких ключевых компонентов, обеспечивающих её функционирование и взаимодействие между пользователями и инфраструктурой. Центральным элементом является мобильное приложение для Android, которое предоставляет пользователям интерфейс для поиска и бронирования зарядных станций, отслеживания процесса зарядки.

Серверная часть системы играет важную роль в обработке и хранении данных. На сервере локально размещается реляционная база данных, которая хранит информацию о пользователях, станциях, коннекторах, заказах и других связанных сущностях. Эта база данных обеспечивает надёжное хранение данных, поддержку различных запросов и возможность их быстрого извлечения по запросу мобильного приложения.

Для связи между мобильным приложением и сервером используется программный интерфейс приложения (API) на базе протокола HTTP, что эффективное взаимодействие между этими компонентами. API позволяет передавать данные в режиме реального времени, обеспечивая синхронизацию всех операций, выполняемых пользователями через приложение, с состоянием системы на сервере. Такая архитектура гарантирует, что пользователи всегда получают актуальную информацию о доступности станций и могут оперативно совершать необходимые действия, такие как бронирование и управление процессом зарядки.

## Описание базы данных

## Модель «сущность-связь»

Для реализации функционала мобильного приложения, связанного с управлением зарядными станциями, была спроектирована база данных, основанная на модели «сущность-связь» (ER-модель). Эта модель служит основой для структурирования данных и позволяет наглядно представить взаимосвязи между основными элементами системы, такими как пользователи, зарядные станции, коннекторы и заказы.

Проектирование ER-модели началось с анализа предметной области и определения ключевых сущностей, необходимых для полноценного функционирования системы. Были учтены все важные аспекты взаимодействия между пользователями и зарядными станциями, включая регистрационные данные, информацию о типах зарядок, процесс создания и выполнения заказов на зарядку.

Одним из важных этапов проектирования было обоснование выбора кардинальных чисел связей между сущностями. Это позволило определить типы связей, такие как «один ко многим» или другие, что играет ключевую роль в обеспечении целостности данных и корректности выполнения запросов. Выбор кардинальности основывался на логике взаимодействия между сущностями.

Результатом проектирования стала ER-модель, представленная на рисунке 1, которая наглядно демонстрирует структуру базы данных и связи между её основными компонентами. Эта модель является фундаментом для дальнейшего создания реляционной базы данных, которая будет использоваться в серверной части для хранения и обработки данных.

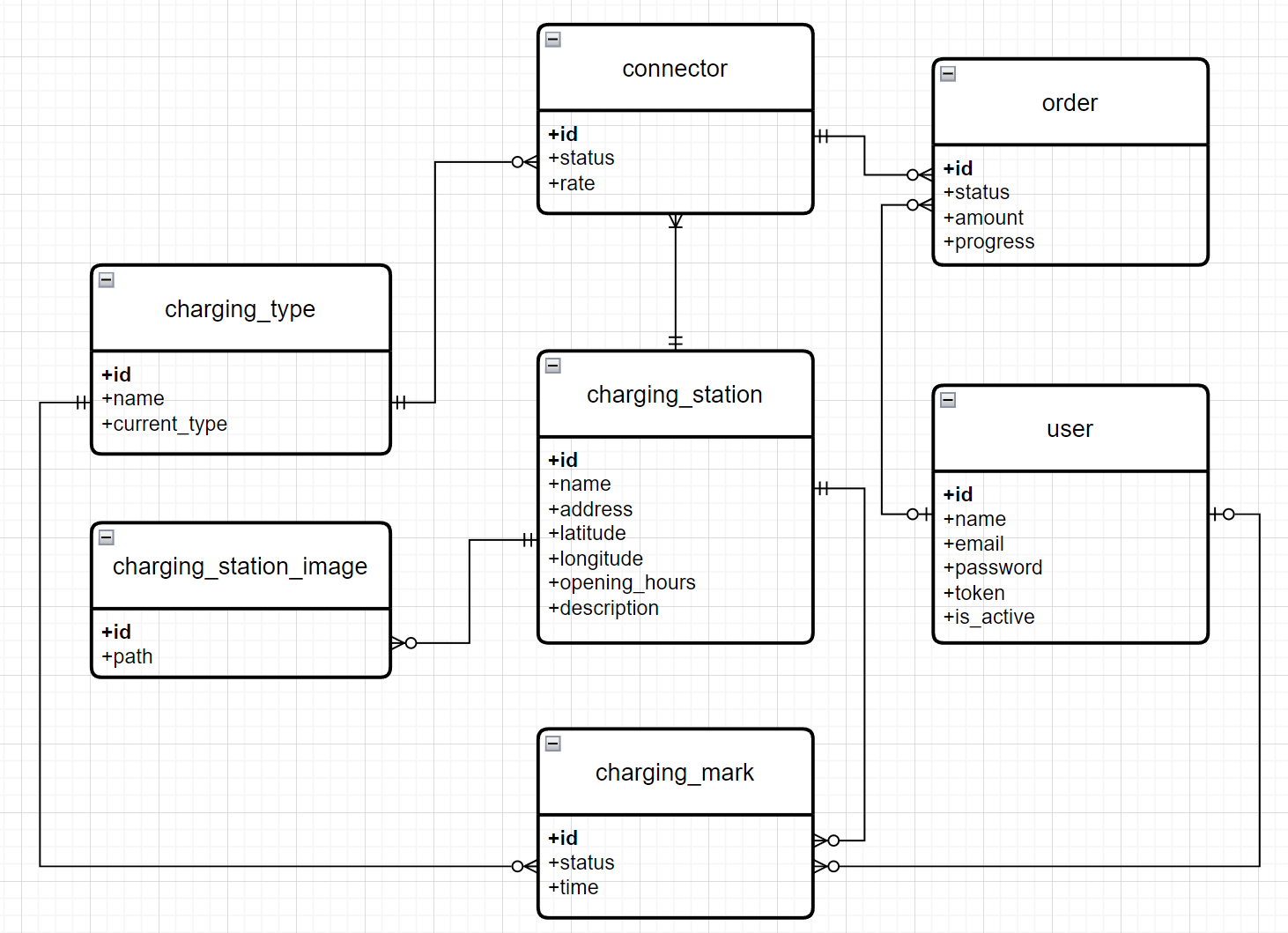


Рисунок 1 – модель «сущность-связь»

Ниже представлено подробное описание модели, включая детальное рассмотрение каждой сущности, ее атрибутов, а также связей между сущностями.

charging\_station (зарядная станция):

* id – идентификатор зарядной станции;
* name – имя зарядной станции;
* address – адрес зарядной станции;
* latitude – широта расположения зарядной станции;
* longitude – долгота расположения зарядной станции;
* opening\_hours – часы работы зарядной станции;
* description – описание зарядной станции;

connector (коннектор):

* id – идентификатор коннектора;
* status – состояние коннектора;
* rate – скорость зарядки коннектора;

order (заказ):

* id – идентификатор заказа;
* status – состояние заказа;
* amount – объем зарядки заказа;
* progress – прогресс заказа;

user (пользователь):

* id – идентификатор пользователя;
* name – имя пользователя;
* email – адрес электронной почты пользователя;
* password – хешированный пароль пользователя;
* token – токен пользователя;
* is\_active – состояние подтверждения пользователя;

charging\_mark (отметка пользователя по зарядке):

* id – идентификатор отметки;
* status – результат зарядки;
* time – время создание отметки;

charging\_station\_image (изображение зарядной станции):

* id – идентификатор изображения;
* path – относительный путь хранения изображения;

charging\_type (тип зарядки):

* id – идентификатор типа зарядки;
* name – название типа зарядки;
* current\_type – тип тока зарядки;

Далее представлено описание связей и обоснование выбора кардинальных чисел.

Связь между сущностями charging\_station и connector характеризуется следующим образом: (1, 1) – (1, N). Это означает, что каждая зарядная станция должна иметь хотя бы 1 связанный с ней коннектор, в то время как каждый коннектор должен быть обязательно привязан ровно к одной станции, что логично, так как коннектор физически устанавливается на конкретной станции и не может существовать отдельно от неё.

Связь между сущностями charging\_station и charging\_station\_image представлена как (1, 1) – (0, N). Это означает, что каждая зарядная станция может иметь одно или несколько изображений, либо не иметь изображений вообще, тогда как каждое изображение обязано быть связано с одной конкретной станцией.

Связь между сущностями charging\_station и charging\_mark имеет кардинальность (1, 1) – (0, N): каждая зарядная станция может иметь ноль или более отметок по зарядке на ней, в то время как каждая отметка привязана ровно к одной конкретной станции.

Связь между сущностями charging\_type и connector характеризуется следующим образом: (1, 1) – (0, N). Это означает, что у каждого типа зарядки могут быть коннекторы с данным типом зарядки или не быть вовсе, а каждому коннектору соответствует в точности один тип зарядки.

Связь между сущностями charging\_type и charging\_mark имеет кардинальность (1, 1) – (0, N): тип зарядки может иметь несколько отметок по зарядке или не иметь, однако каждая отметка должна быть привязана ровно к одному типу зарядки.

Связь между сущностями connector и order представлена как (1, 1) – (0, N). Это означает, что каждому коннектору может соответствовать один или несколько заказов, либо не соответствовать вообще, а каждый заказ должен быть связан только с одним коннектором, поскольку зарядка происходит только от него самого.

Связь между сущностями user и order характеризуется следующим образом: (0, 1) – (0, N). Это означает, что каждый пользователь может иметь один или несколько заказов, а также не иметь их вовсе, в то время как каждый заказ может принадлежать одному пользователю, либо не принадлежать, так как пользователь может быть неавторизованным.

Наконец, связь между сущностями user и charging\_mark имеет кардинальность (0, 1) – (0, N): каждому пользователю может соответствовать ноль или больше отметок по зарядке, тогда как каждая отметка может быть связана только с одним пользователем, либо не связана совсем, поскольку пользователь аналогично предыдущей связи может быть неавторизованным и оставить отметку.

## Реляционная модель

Для преобразования ER-модели в реляционную модель существует ряд шагов, каждый из которых направлен на то, чтобы отразить структуру и логику исходной модели в терминах реляционной базы данных.

Первый этап заключается в создании таблиц для каждой сущности ER-модели. На этом шаге происходит определение первичных ключей для каждой таблицы. Чаще всего, такие ключи являются суррогатными, то есть специально созданными уникальными идентификаторами, не несущими смысловой нагрузки, но предназначенными для обеспечения уникальности каждой записи. Помимо этого, для каждого столбца таблицы определяется его тип данных, возможность наличия пустых значений (NULL), значения по умолчанию и различные ограничения, которые могут быть наложены на допустимые значения данных.

Второй шаг связан с установлением связей между сущностями, которые реализуются через внешние ключи. Внешний ключ создается в таблице, представляющей одну сущность, и ссылается на первичный ключ другой сущности, отражая связи, существующие между ними в ER-модели.

Третий этап посвящен обеспечению условий минимальной кардинальности, то есть определению обязательности связей между сущностями. В зависимости от конкретных условий, сущности могут быть обязательными (mandatory) или необязательными (optional) по отношению друг к другу. На основании этого выделяют четыре варианта минимальной кардинальности: обязательная и необязательная сущность с обеих сторон связи (optional-optional), обязательная с одной стороны и необязательная с другой (mandatory-optional и optional-mandatory), либо обе сущности являются обязательными (mandatory-mandatory). В зависимости от типа связи накладываются ограничения на операции удаления, обновления и вставки данных для родительских и дочерних сущностей.

При создании таблиц для каждой сущности ER-модели, были выбраны соответствующие первичные ключи, которые представляют собой уникальные числовые значения. Эти ключи являются суррогатными. Все атрибуты сущностей были также описаны с указанием типов данных, допустимости пустых значений и ограничений, что обеспечивает корректность данных в таблицах.

По завершении стало обеспечение условий минимальной кардинальности, которое было достигнуто через наложение соответствующих ограничений на операции удаления, обновления и вставки данных для каждой сущности. В результате всех шагов алгоритма ER-модель была преобразована в реляционную модель, показанная на рисунке 2.

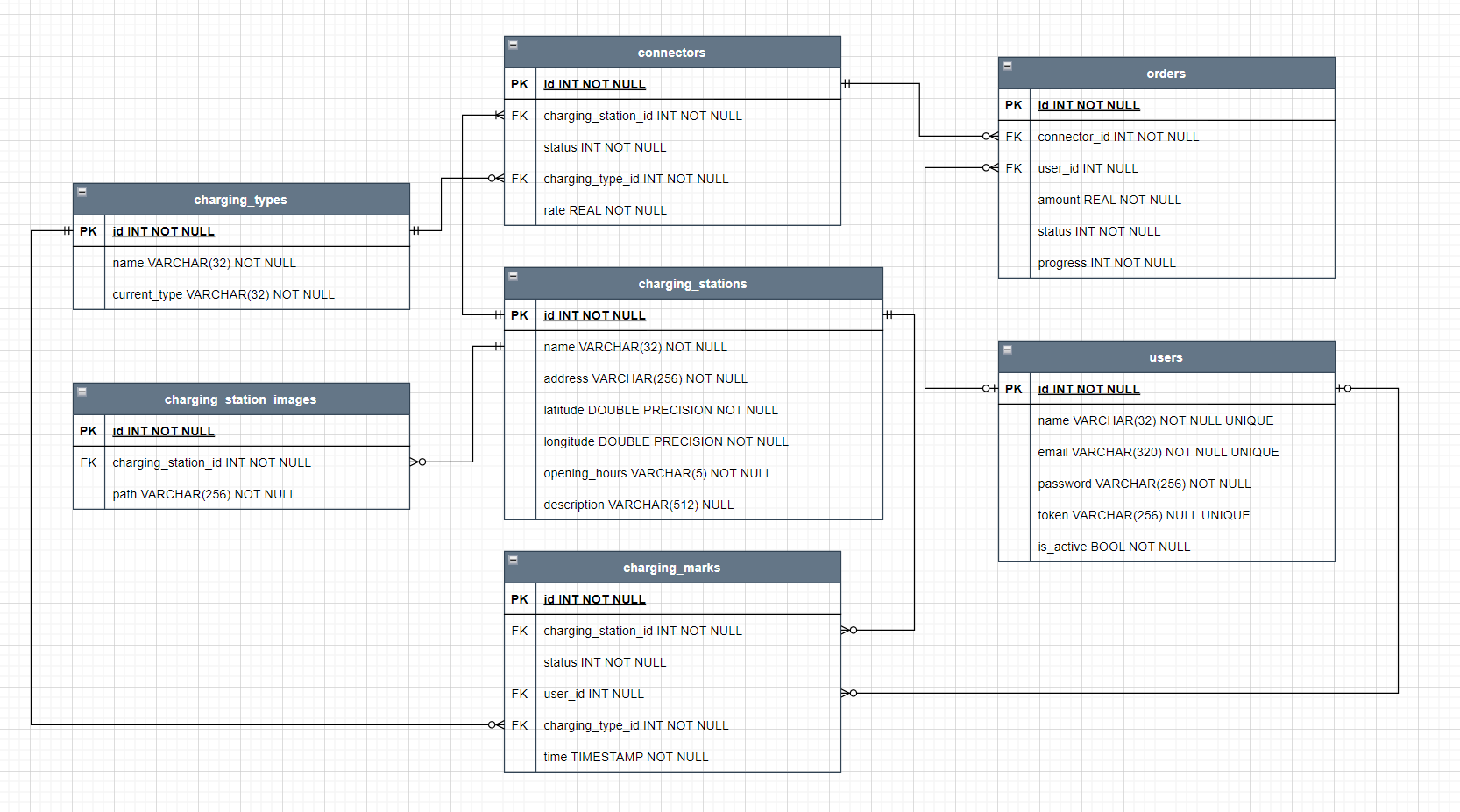


Рисунок 2 – реляционная модель

Обоснование выбора типов данных и ключей представлено в таблицах 1-7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Type | Key | NULL Status | Remarks |
| id | INT | Primary Key | NOT NULL | Surrogate Key |
| name | VARCHAR(32) | No | NOT NULL |  |
| address | VARCHAR(256) | No | NOT NULL |  |
| latitude | DOUBLE  PRECISION | No | NOT NULL |  |
| longitude | DOUBLE  PRECISION | No | NOT NULL |  |
| opening\_hours | VARCHAR(5) | No | NOT NULL | opening\_hours ~ '^\d{1,2}-\d{1,2}$' |
| description | VARCHAR(512) | No | NULL |  |

Таблица 1 – charging\_stations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Type | Key | NULL Status | Remarks |
| id | INT | Primary Key | NOT NULL | Surrogate Key |
| charging\_station\_id | INT | Foreign Key | NOT NULL |  |
| status | INT | No | NOT NULL |  |
| charging\_type\_id | INT | Foreign Key | NOT NULL |  |
| rate | REAL | No | NOT NULL | rate > 0 |

Таблица 2 – connectors

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Type | Key | NULL Status | Remarks |
| id | INT | Primary Key | NOT NULL | Surrogate Key  IDENTITY(1,1) |
| charging\_station\_id | INT | Foreign Key | NOT NULL |  |
| path | VARCHAR(256) | No | NOT NULL |  |

Таблица 3 – charging\_station\_images

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Type | Key | NULL Status | Remarks |
| id | INT | Primary Key | NOT NULL | Surrogate Key  IDENTITY(1,1) |
| charging\_station\_id | INT | Foreign Key | NOT NULL |  |
| status | INT | No | NOT NULL |  |
| user\_id | INT | Foreign Key | NULL |  |
| time | TIMESTAMP | No | NOT NULL |  |

Таблица 4 – charging\_marks

Сущность charging\_stations (зарядная станция) имеет три взаимосвязанные дочерние сущности: connectors (коннектор), charging\_station\_images (изображение зарядной станции) и charging\_marks (отметка по зарядке). Для первой связи установлена минимальная кардинальности типа mandatory-mandatory, а для остальных связей - минимальная кардинальности типа mandatory-optional. Последнее означает, что каждая дочерняя сущность должна быть связана с конкретной зарядной станцией, в то время как для самой станции наличие дочерних сущностей не является обязательным условием.

При создании зарядной станции должен существовать хотя бы один связанный с ней коннектор. Кроме того, при создании любой из дочерних сущностей, таких как коннекторы, изображения или отметки, необходимо указать связь с соответствующей зарядной станцией.

Если же зарядную станцию потребуется удалить, то связанные с ней дочерние сущности будут автоматически удалены благодаря механизму каскадного удаления. Это гарантирует целостность данных и предотвращает ситуации, когда дочерние записи остаются в базе без родительской сущности. В то же время на удаление дочерних сущностей ограничений нет — они могут быть удалены без влияния на родительскую сущность, кроме сущности connectors, где удаление запрещено в том случае, если остался последний связанный с конкретной зарядной станцией коннектор. Что касается обновления ключей, то любые изменения первичных и внешних ключей запрещены для всех сущностей, что помогает сохранить структуру и связи между сущностями неизменными.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Type | Key | NULL Status | Remarks |
| id | INT | Primary Key | NOT NULL | Surrogate Key  IDENTITY(1,1) |
| name | VARCHAR(32) | No | NOT NULL |  |
| current\_type | VARCHAR(32) | No | NOT NULL |  |

Таблица 5 – charging\_types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Type | Key | NULL Status | Remarks |
| id | INT | Primary Key | NOT NULL | Surrogate Key  IDENTITY(1,1) |
| connector\_id | INT | Foreign Key | NOT NULL |  |
| user\_id | INT | Foreign Key | NULL |  |
| amount | REAL | No | NOT NULL | amount > 0 |
| status | INT | No | NOT NULL |  |
| progress | INT | No | NOT NULL | DEFAULT value = 0  progress >= 0 AND  progress <= 100 |

Таблица 6 – orders

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column Name | Type | Key | NULL Status | Remarks |
| id | INT | Primary Key | NOT NULL | Surrogate Key  IDENTITY(1,1) |
| name | VARCHAR(32) | No | NOT NULL |  |
| email | VARCHAR(320) | No | NOT NULL | UNIQUE  (AK 1.1) |
| password | VARCHAR(256) | No | NOT NULL |  |
| token | VARCHAR(256) | No | NULL | UNIQUE  (AK 2.1) |
| is\_active | BOOLEAN | No | NOT NULL | DEFAULT value = false |

Таблица 7 – users

Сущность connectors связана с дочерней сущностью orders (заказы) через механизм каскадного удаления. Это означает, что при удалении записи о конкретном коннекторе все заказы, которые были связаны с этим коннектором, автоматически удаляются из системы. Такой подход позволяет поддерживать целостность данных, исключая ситуации, когда заказы остаются в базе данных без привязки к соответствующему коннектору.

Что касается сущности charging\_types (тип зарядки), то она обладает более строгими правилами удаления. Ее нельзя удалить напрямую, если существуют дочерние записи, связанные с ней. Перед удалением charging\_types необходимо вручную разорвать все связи с дочерними сущностями, чтобы система позволила выполнить удаление. Это предотвращает удаление данных, которые могут быть критичны для других зависимых записей.

При удалении сущности users (пользователь) каскадно удаляются все связанные записи из charging\_marks (отметка по зарядке) и orders (заказ). Это гарантирует, что при удалении пользователя из системы все его заказы и отметки по зарядке также удаляются.

## Описание API взаимодействия приложения с сервером

API использует протокол HTTP для передачи данных, и вся коммуникация между клиентом и сервером осуществляется через отправку HTTP-запросов и получение соответствующих HTTP-ответов, тело которых представлено в формате JSON.

Каждый запрос содержит определённые параметры и данные, необходимые для выполнения операции, такие как авторизация пользователя, получение информации о зарядных станциях и создание заказов. В ответ на запросы сервер возвращает данные, соответствующие запрашиваемой информации, или подтверждение успешного выполнения операции. В том случае, если запрос не удалось обработать, то по умолчанию будет возвращен ответ с кодом 500, что означает ошибку. Далее будут представлены описание основных запросов с примерами.

## Аутентификация пользователя

Аутентификация пользователя осуществляется с помощью POST запроса, отправляемого на конечную точку /auth. Этот запрос предназначен для проверки учетных данных пользователя по почте и паролю, а также получения токена.

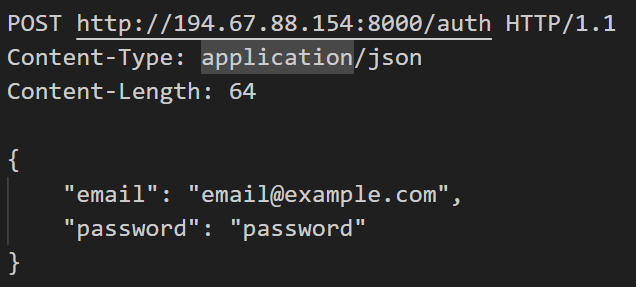


Рисунок 3 – пример запроса аутентификации пользователя

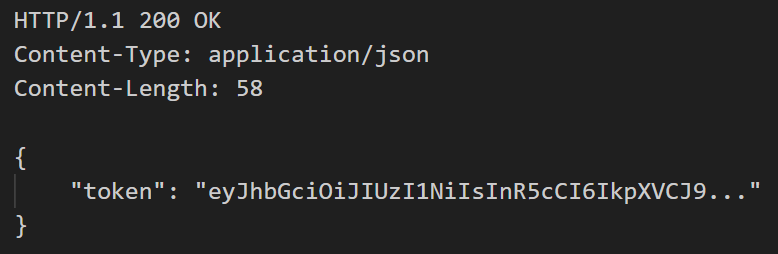


Рисунок 4 – пример ответа аутентификации пользователя

## Регистрация пользователя

Регистрация нового пользователя осуществляется с помощью POST запроса, отправляемого на конечную точку /register. Этот запрос используется для создания новой учетной записи, используя имя, адрес электронной почты и пароль, при этом сервер проверяет, свободен ли указанный адрес электронной почты. Если регистрация прошла успешно, то на указанный адрес электронной почты отправляется письмо со ссылкой для подтверждения учетной записи. Пользователь должен перейти по этой ссылке, чтобы завершить процесс регистрации и активировать свою учетную запись.

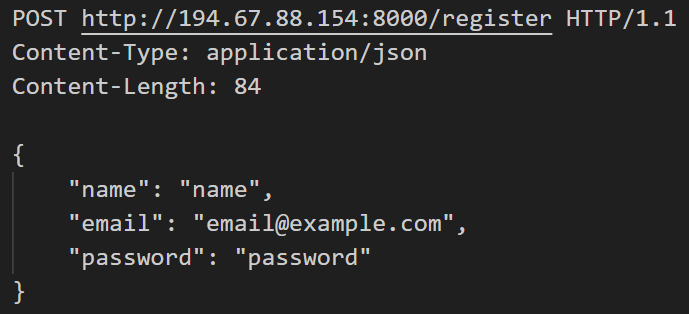


Рисунок 5 – пример запроса регистрации пользователя

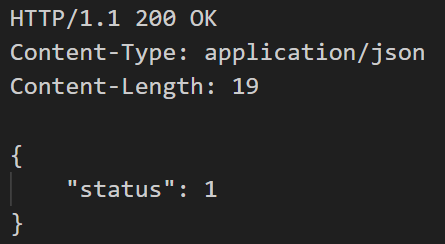


Рисунок 6 – пример ответа регистрации пользователя

## Получения списка зарядных станций

Для получения списка зарядных станций используется GET запрос к конечной точке /charging-stations. Этот запрос позволяет получить информацию о существующих зарядных станциях с возможностями фильтрации по уровню детализации и по подстроке адреса, которым соответствуют параметры level и query соответственно.

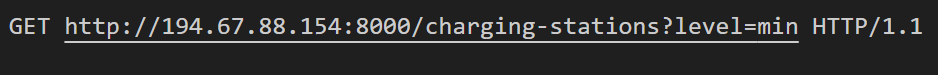


Рисунок 7 – пример запроса получения списка зарядных станций

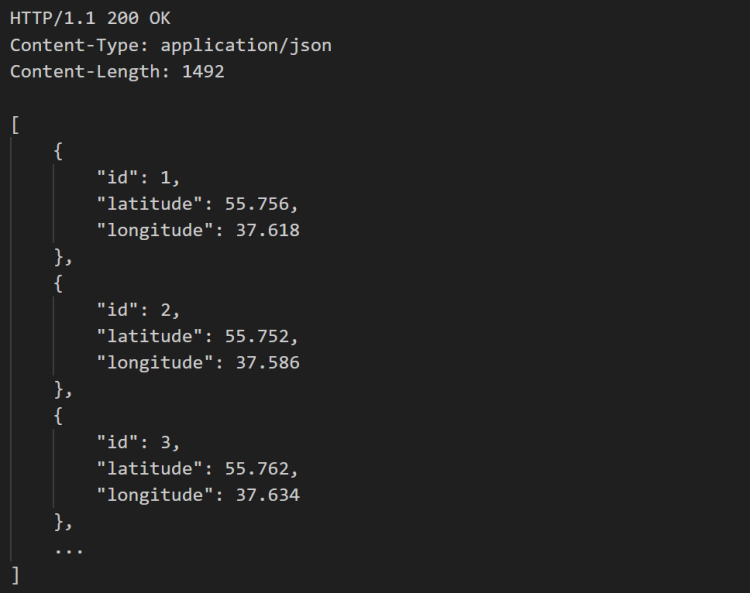


Рисунок 8 – пример ответа получения списка зарядных станций

## Получение подробной информации о зарядной станции

Получение подробной информации о зарядной станции осуществляется с помощью GET запроса, отправляемого на конечную точку /charging-stations/{id}, где {id} – это идентификатор станции.

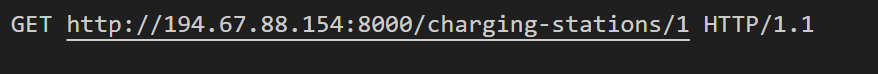


Рисунок 9 – пример запроса получения подробной информации о зарядной станции

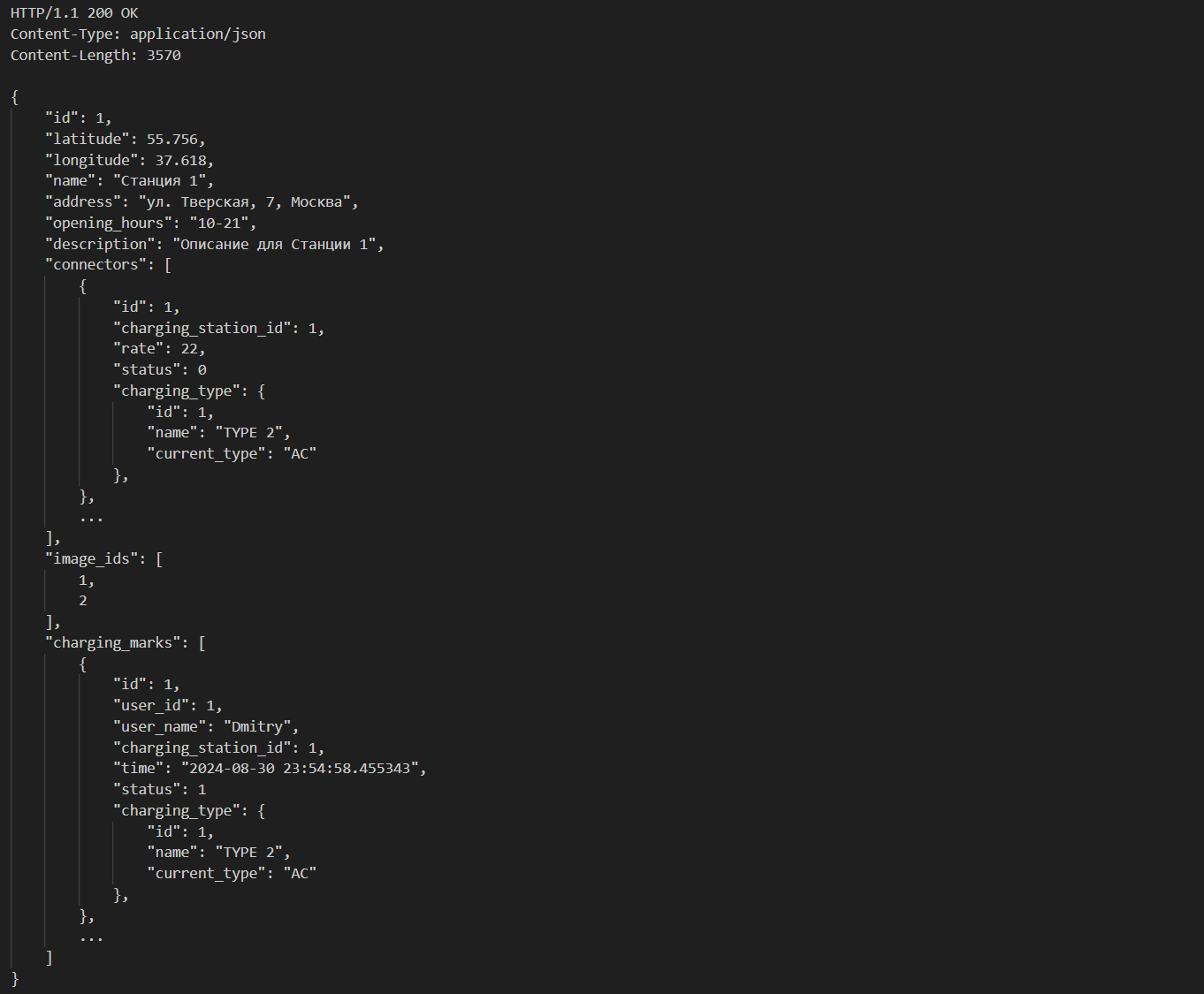


Рисунок 10 – пример ответа получения подробной информации о зарядной станции

## Создание заказа

Создание заказа осуществляется с помощью POST запроса, отправляемого на конечную точку /charge. Этот запрос используется для создания нового заказа, используя идентификатор коннектора, объем зарядки и токен пользователя, если он авторизован. В случае успеха в ответе вернется идентификатор заказа.

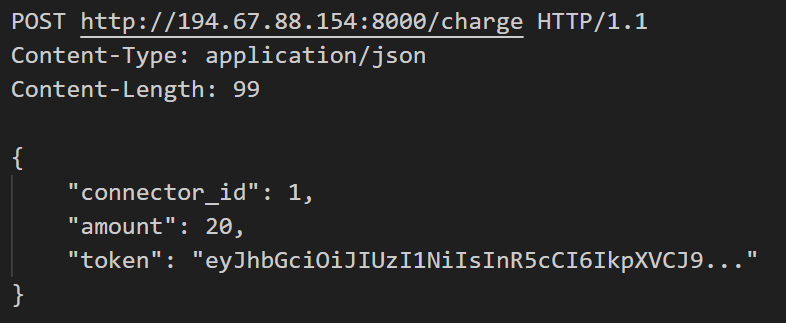


Рисунок 11 – пример запроса создания заказа

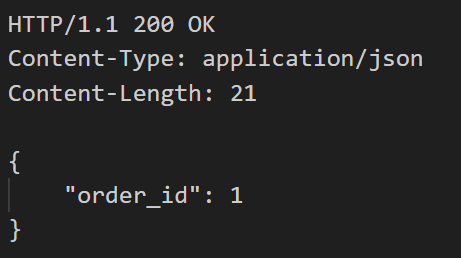


Рисунок 12 – пример ответа создания заказа

## Получение информации по заказу

Для получения информации по заказу используется GET запрос к конечной точке /orders/{id}, где {id} – это идентификатор заказа.



Рисунок 13 – пример запроса получения информации по заказу

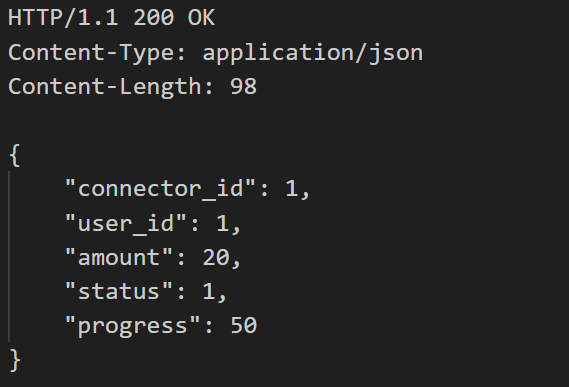


Рисунок 14 – пример ответа получения информации по заказу

## Создание отметки по зарядке

Создание отметки осуществляется с помощью POST запроса, отправляемого на конечную точку /mark. Этот запрос используется для создания новой отметки после завершения заказа, где пользователь может указать о том, как прошла зарядка.

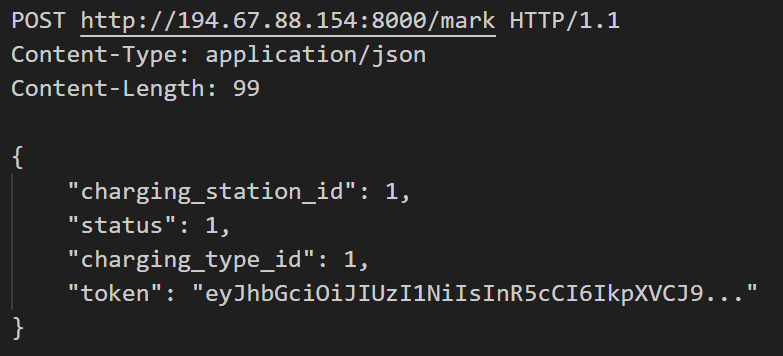


Рисунок 15 – пример запроса создания отметки по заказу



Рисунок 16 – пример ответа создания отметки по заказу

## Подтверждение учетной записи

Для подтверждения учетной записи используется GET запрос к конечной точке /confirm. Этот запрос позволяет подтвердить учетную запись по токену, который содержится в составе ссылки, которая приходит на адрес электронной почты, указанной при регистрации.

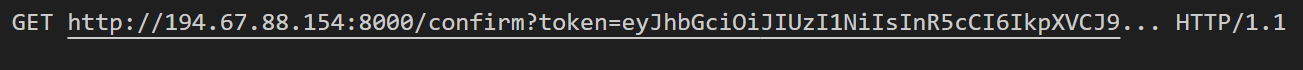


Рисунок 17 – пример запроса подтверждения учетной записи

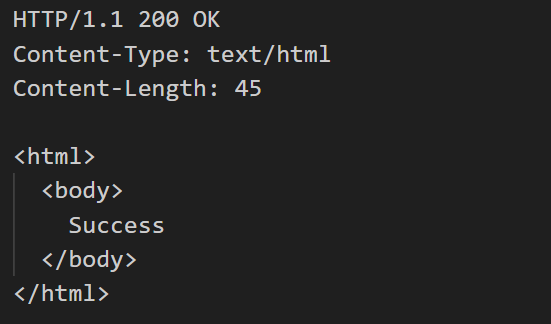


Рисунок 18 – пример ответа подтверждения учетной записи

## Получение изображения зарядной станции

Получение изображения зарядной станции осуществляется с помощью GET запроса, отправляемого на конечную точку /charging-stations-images/{id}, где {id} – это идентификатор изображения.

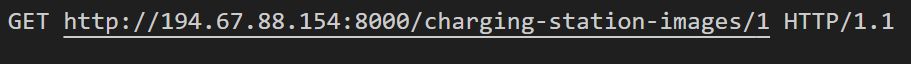


Рисунок 19 – пример запроса учетной записи

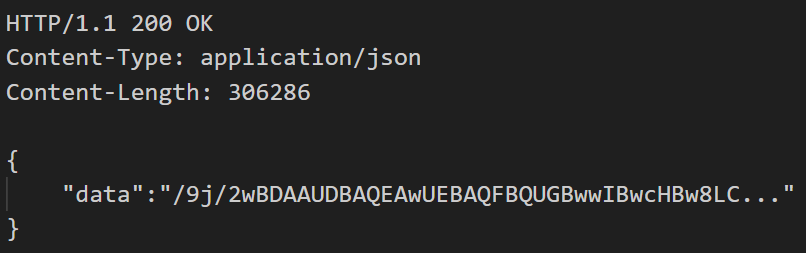


Рисунок 20 – пример ответа подтверждения учетной записи

## Стек технологий

Для реализации мобильного приложения и серверной части был использован широкий спектр современных технологий. Эти технологии обеспечивают высокую производительность, безопасность и удобство использования, а также облегчают разработку, тестирование и поддержку системы.

Мобильное приложение для платформы Android было разработано с использованием языка программирования Kotlin. Этот язык предоставляет современные возможности и синтаксические улучшения по сравнению с Java, что позволяет писать более выразительный и безопасный код. Для управления зависимостями и сборки проекта использовалась система Gradle, которая упрощает процесс интеграции библиотек и управления проектом.

Основой пользовательского интерфейса приложения является Jetpack Compose — современная декларативная библиотека для создания интерфейсов на Android. Использование Jetpack Compose позволило существенно упростить процесс разработки интерфейсов, сделав его более гибким и удобным.

Для отображения карт и интеграции геолокационных данных в приложении использовался MapKit от Яндекс. Эта библиотека позволяет легко добавлять карты, работать с географическими координатами и прокладывать маршруты к зарядным станциям.

Для взаимодействия с сервером и обработки сетевых запросов был использован Retrofit — популярная библиотека, которая упрощает работу с REST API и позволяет эффективно управлять запросами и ответами.

Для работы с QR-кодами в приложении была использована библиотека ZXing, которая обеспечивает быструю и точную обработку QR-кодов, что является важным для процесса идентификации зарядных станций.

Архитектура приложения построена на основе Clean Architecture с использованием паттерна MVVM (Model-View-ViewModel). Такой подход разделяет логику приложения на отдельные слои, что упрощает поддержку и тестирование, а также делает код более структурированным и масштабируемым.

Серверная часть системы была реализована на языке программирования Java, что обеспечивает высокую производительность и надежность при работе с большими объемами данных. Для управления зависимостями и сборки проекта использовалась система Maven.

Для работы с базой данных использовался JDBC (Java Database Connectivity) — стандартный API для взаимодействия с реляционными базами данных, который обеспечивает надежное и быстрое подключение к базе данных и выполнение SQL-запросов. В качестве базы данных была выбрана PostgreSQL — мощная и расширяемая реляционная СУБД, обеспечивающая высокий уровень производительности и безопасности данных.

Для сериализации и десериализации данных в формате JSON использовалась библиотека Gson, что упрощает обмен данными между клиентом и сервером.

Для управления электронной почтой применялась библиотека com.sun.mail, которая поддерживает протоколы SMTP, IMAP и POP3 и позволяет легко интегрировать функционал отправки писем.

## Реализация

Для реализации серверной части был выделен сервер на базе операционной системы Linux с достаточными вычислительными и сетевыми ресурсами для обеспечения корректной и бесперебойной работы. Сервер располагается в выделенной среде и имеет доступ к необходимым компонентам и библиотекам для выполнения задач.

На сервере, в директории /root, была создана папка chargingstations, которая служит основным хранилищем всех файлов и данных, необходимых для работы серверной части. Внутри этой директории размещен исполняемый файл .jar, который используется для запуска серверного приложения, который запускается

Кроме того, в папке chargingstations находится файл email.txt, который содержит конфиденциальную информацию, необходимую для отправки электронных писем с сервера. Этот файл включает данные для авторизации на почтовом сервер, что является важной частью взаимодействия с пользователями в рамках данного проекта.

Также в директории chargingstations была создана папка images, в которой хранятся изображения зарядных станций.

HTTP сервер реализован с помощью класса HttpServer. Сервер привязан к IP-адресу 0.0.0.0 и порту 8000, что позволяет принимать запросы на всех интерфейсах. Для управления потоками используется встроенный механизм, так как Executor не был задан (установлено null). После этого сервер запускается с выводом подтверждения успешного старта. На рисунке 21 показан процесс инициализации и запуска сервера.

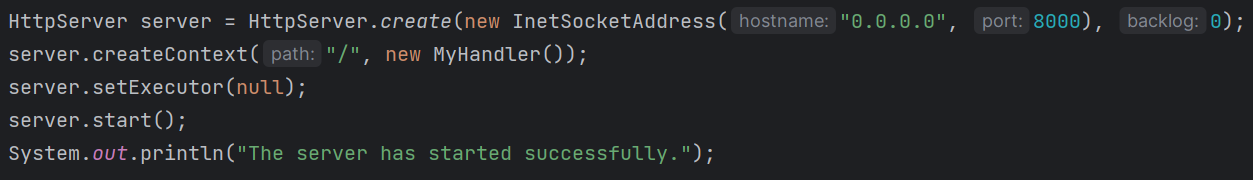


Рисунок 21 – процесс инициализации и запуска сервера

Запросы на сервере обрабатываются с использованием регулярных выражений для сопоставления URI запроса с определенными шаблонами. В случае аутентификации пользователя шаблон запроса определяется как /auth. Если поступивший запрос соответствует этому шаблону и является GET запрасом, начинается процесс обработки данных для проверки учетных записей. Код обработки приведен на рисунке 22.

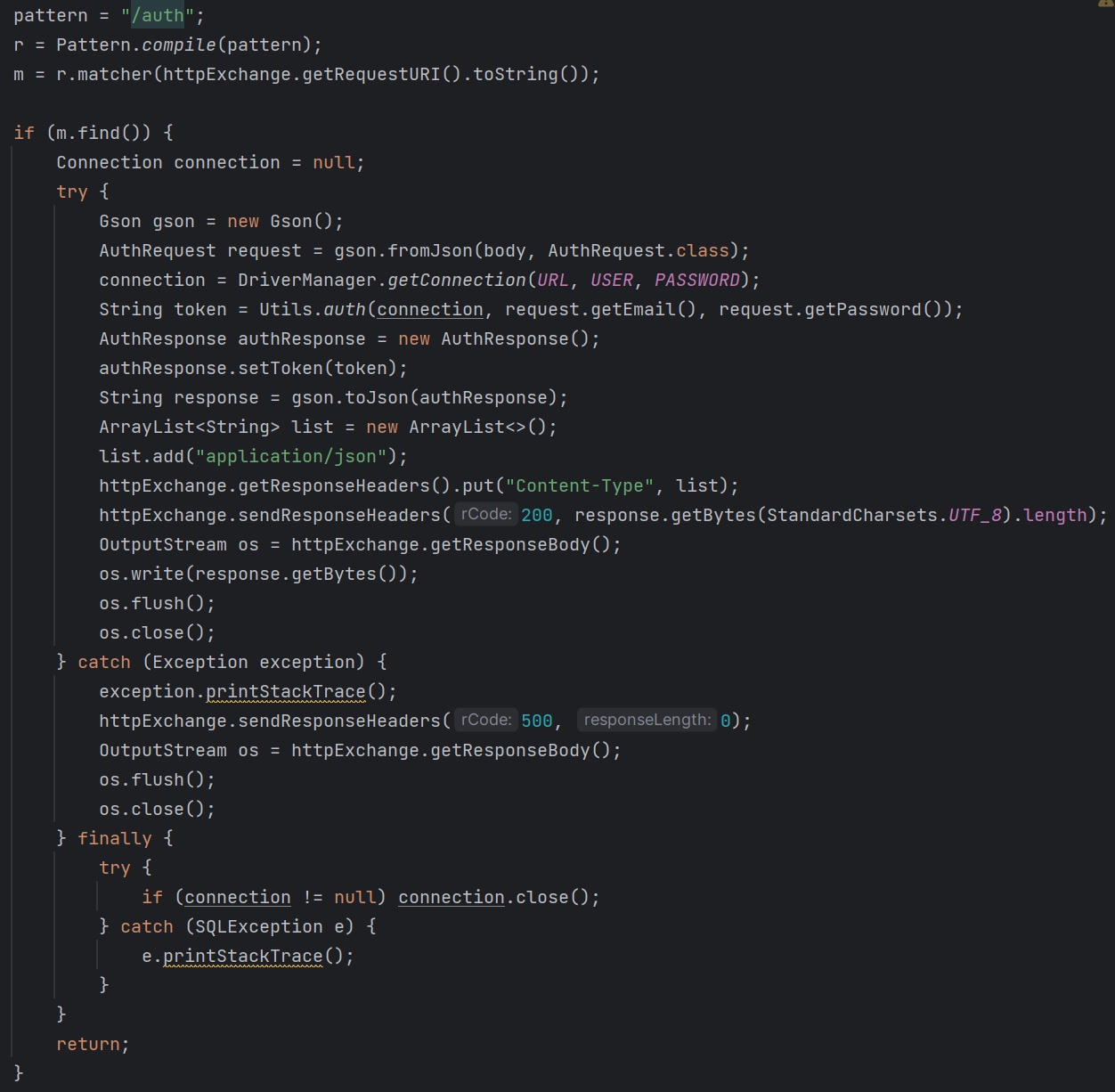


Рисунок 22 – обработка запроса аутентификации пользователя

Сначала из тела запроса извлекаются данные, такие как email и пароль. Для этого используется библиотека Gson, которая преобразует полученные данные из формата JSON в объект, с которым сервер может работать. После извлечения данных создается соединение с базой данных с использованием параметров подключения (адрес базы данных, имя пользователя и пароль). Это соединение необходимо для того, чтобы проверить, есть ли в базе данных указанный пользователь и верный ли у него пароль.

Если учетные данные пользователя совпадают с данными в базе, вызывается функция, которая возвращает токен. Она принимает в качестве аргументов соединение с базой данных, адрес электронной почты и пароль пользователя. Работа функции разделена на несколько этапов. Код данной функций представлен на рисунке 23.

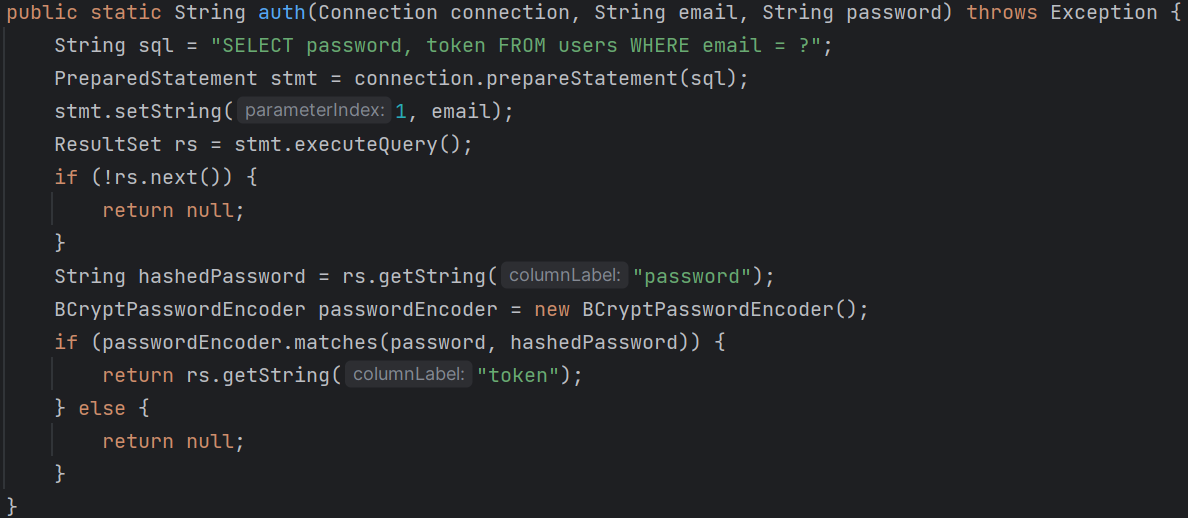


Рисунок 23 – функция аутентификации пользователя

Сначала формируется SQL-запрос, который ищет пользователя по email. В запросе используется параметризированный SQL для предотвращения SQL-инъекций. После подготовки запроса с помощью объекта PreparedStatement, в него вставляется адрес электронной почты, который был получен от пользователя, и запрос выполняется.

Результат выполнения запроса помещается в объект ResultSet, который содержит данные, полученные из базы. Если в базе данных нет пользователя с указанным адресом электронной почты, то есть ResultSet пустой, функция возвращает null, указывая, что аутентификация неуспешна.

Если же пользователь с таким адресом электронной почты найден, функция извлекает из базы два поля: password и token. Пароль хранится в зашифрованном виде (хешированный пароль), поэтому для проверки подлинности введенного пароля используется объект BCryptPasswordEncoder. Этот объект проверяет, соответствует ли введенный пароль его хешированной версии, сохраненной в базе данных.

Если введенный пароль совпадает с хешем, который хранится в базе, функция возвращает токен пользователя, связанный с этим пользователем. В случае, если пароли не совпадают, функция также возвращает null, сигнализируя о неудачной аутентификации.

Затем этот токен упаковывается в ответное сообщение, и сервер отправляет его клиенту. Перед отправкой устанавливаются необходимые заголовки, например, тип содержимого ("application/json"), чтобы клиент правильно интерпретировал ответ. После этого сервер отправляет ответ с кодом 200, подтверждающим успешную аутентификацию.

В случае возникновения ошибки в процессе аутентификации, например, при проблемах с базой данных или неверных данных, сервер возвращает код ошибки 500. При этом ресурсы, такие как соединение с базой и потоки данных, закрываются в любом случае, чтобы избежать утечек и сбоев.

Класс EmailSender был разработан для отправки сообщений на почту, показанный на рисунке 24, где содержится ссылка, по нажатию которой происходит активация учетной записи пользователя.



Рисунок 24 – класс EmailSender

Для инициализации базы данных, которая хранится на сервере, был разработан и использован скрипт init.sql, который играет ключевую роль в установке и настройке базы данных. Этот скрипт представляет собой комплексное решение, включающее создание таблиц, определение функций и настройку триггеров, что позволяет обеспечить корректное функционирование и целостность базы данных.

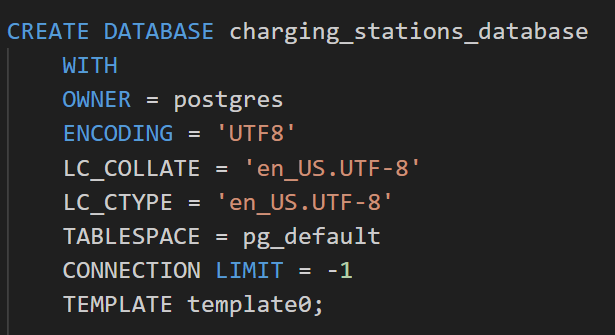


Рисунок 25 – фрагмент init.sql (создание базы данных)

После определения базы данных было осуществлено создание таблиц, что является основой для хранения данных. Каждая таблица была спроектирована с учетом структуры данных и связей между сущностями, что позволяет создать надежный фундамент для базы данных. Эти таблицы включают в себя необходимые поля и типы данных, а также уникальные ограничения, которые гарантируют правильность хранения информации, на основе реляционной модели.

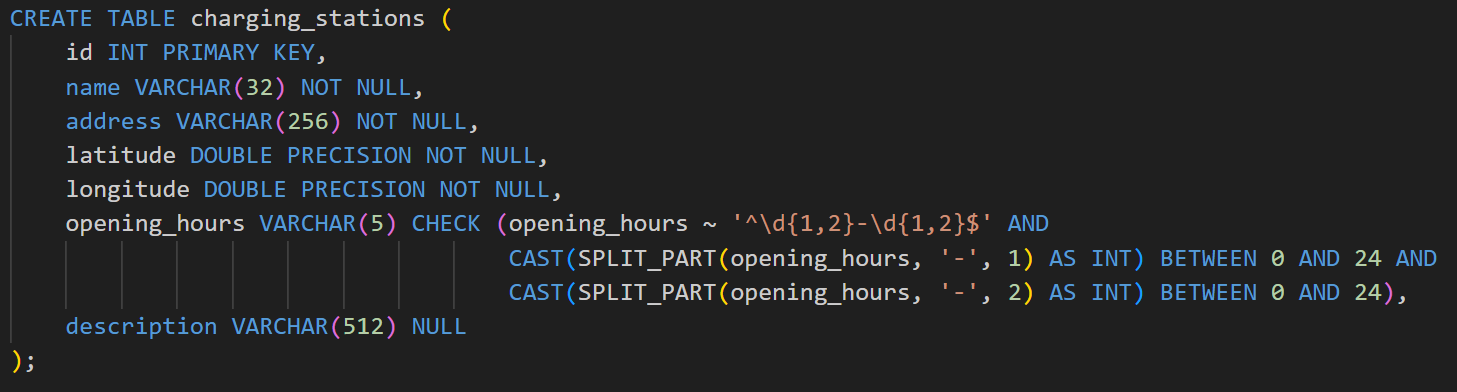


Рисунок 26 – фрагмент init.sql (создание одной из таблиц)

Кроме того, скрипт содержит определение функций, которые играют важную роль в поддержании логики работы базы данных. Эти функции обеспечивают выполнение специальных проверок и операций, необходимых для обеспечения целостности данных и корректного функционирования системы.

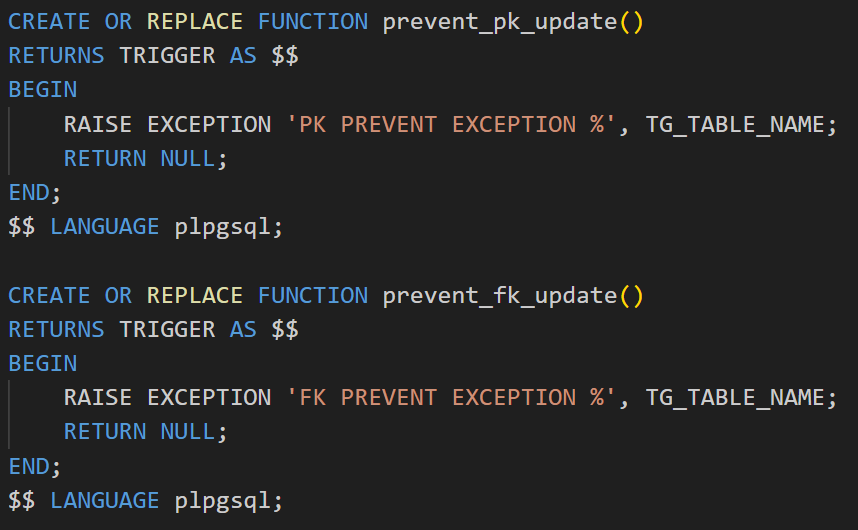


Рисунок 27 – фрагмент init.sql (создание функций)

Особое внимание было уделено настройке триггеров. Они были созданы для автоматического контроля и управления изменениями в базе данных. Каждый триггер настроен так, чтобы срабатывать при выполнении определенных операций, таких как обновление данных в таблицах. Эти триггеры помогают предотвращать нарушения целостности данных, обеспечивая тем самым, что все связи между таблицами остаются корректными и непротиворечивыми.

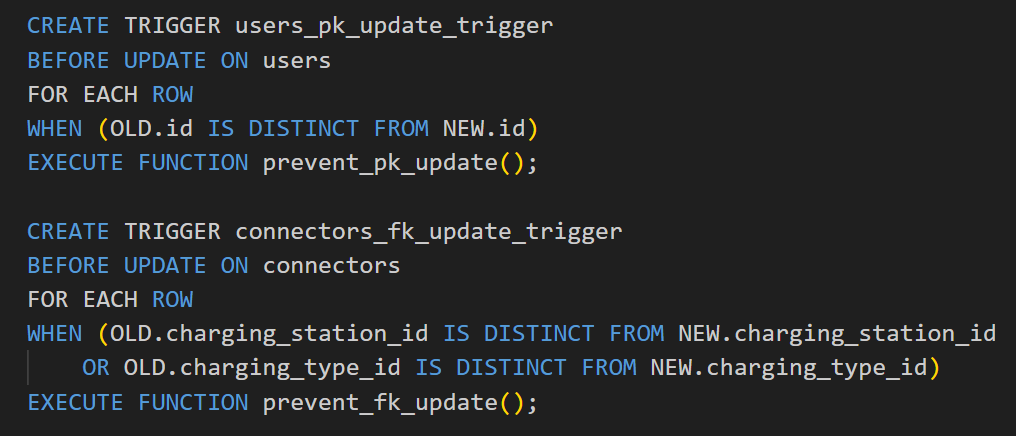


Рисунок 28 – фрагмент init.sql (создание триггеров)

Мобильное приложение было разработано с применением архитектурного паттерна MVVM (Model-View-ViewModel), что позволило достичь четкого разделения ответственности между компонентами, обеспечив при этом модульность, гибкость и упрощение процесса тестирования. Использование этого паттерна предоставляет возможность отделить логику приложения от его пользовательского интерфейса, что способствует улучшению качества кода и его последующей поддерживаемости.

В процессе разработки были созданы различные активности, такие как SplashActivity, MainActivity, AuthenticationActivity и OrderActivity. Каждая из них отвечает за конкретные задачи и обеспечивает соответствующий функционал приложения. Например, SplashActivity была реализована как стартовый экран, отображаемый при запуске приложения. MainActivity представляет собой основной экран, где пользователю предоставляется доступ к основным функциям приложения, и за счет использования ViewModel обеспечивается взаимодействие с бизнес-логикой и отображение данных на экране.

AuthenticationActivity отвечает за функционал авторизации и регистрации пользователей, предоставляя удобный интерфейс для ввода логина и пароля. Взаимодействие с сервером, а также проверка введенных данных происходят через соответствующий ViewModel. OrderActivity реализует логику работы с заказами. Здесь также ViewModel играет ключевую роль в обеспечении взаимодействия между интерфейсом и данными.

Каждой активности соответствует свой ViewModel, что позволяет отделить управление данными от представления интерфейса. ViewModel обрабатывает все необходимые запросы, взаимодействует с моделью данных и передает результаты обратно во View, что обеспечивает плавную и эффективную работу приложения. Такой подход не только улучшает организацию кода, но и позволяет легко добавлять новые функции, тестировать отдельные компоненты, а также снижает риск ошибок, связанных с неправильным взаимодействием между элементами интерфейса и данными.

Также в процессе разработки мобильного приложения была использована современная библиотека для создания пользовательского интерфейса — Jetpack Compose, которая позволяет существенно упростить процесс разработки и сделать его более интуитивно понятным и гибким. Благодаря Compose, графические элементы были структурированы в виде отдельных функций. Это позволило эффективно разделять ответственность между различными частями интерфейса и организовать код так, чтобы он оставался чистым, легко читаемым и поддерживаемым.

Каждая такая функция представляла собой самостоятельный компонент интерфейса, который можно было переиспользовать в разных частях приложения. Это дало возможность организовать разработку таким образом, чтобы сложные экраны и элементы интерфейса состояли из небольших, хорошо определенных и изолированных частей, которые можно было легко тестировать и изменять. Например, элементы управления, формы ввода данных, кнопки и текстовые поля были реализованы как отдельные функции, что упростило их настройку и стилизацию. Такой подход позволил сосредоточиться на создании красивого и отзывчивого интерфейса, при этом значительно сократив количество дублируемого кода.

Помимо работы с элементами пользовательского интерфейса, в приложении также использовалась карта, которая была интегрирована с помощью MapKit. Карта являлась важным элементом интерфейса, так как позволяла пользователям взаимодействовать с географическими данными, будь то отображение местоположения, указание маршрутов или получение информации о конкретных объектах. В рамках реализации работы с картой были созданы отдельные компоуз функции, которые отвечали за ее отрисовку и взаимодействие с пользователем. Это позволило внедрить карту в общий интерфейс приложения, сохранив при этом чистоту кода и соблюдение архитектурных принципов.

Использование MapKit в сочетании с Jetpack Compose дало возможность не только эффективно управлять отображением карты, но и обеспечить плавное взаимодействие с другими элементами интерфейса. Например, элементы управления на карте (приближение, перемещение) были органично интегрированы в структуру функций, что дало возможность пользователям интуитивно и просто взаимодействовать с картой. Также важным аспектом стало управление состоянием карты через ViewModel, что позволило сохранить данные о текущем состоянии карты, такие как масштаб или местоположение, даже при смене ориентации экрана или переходе между экранами.

Для выполнения сетевых запросов в приложении был использован популярный и мощный инструмент — Retrofit, который предоставляет удобный способ работы с REST API. Retrofit позволяет легко взаимодействовать с сервером, отправлять запросы и получать ответы, автоматически преобразуя данные в нужные объекты с помощью конвертеров, таких как Gson. Это делает процесс разработки простым и интуитивным, так как разработчику не приходится вручную обрабатывать JSON-ответы или кодировать данные.

Для взаимодействия с API была создана, изображенная на рисунке 29, интерфейсная реализация ApiService, которая определяет все необходимые запросы, используемые в приложении. Каждый метод интерфейса соответствует определенной конечной точке API и возвращает объект типа Call, который представляет собой запрос, способный выполнить синхронное или асинхронное действие. Благодаря гибкости Retrofit, запросы были настроены с использованием аннотаций, таких как @GET, @POST, @Path и @Query, что делает код максимально читаемым и легким для изменения.

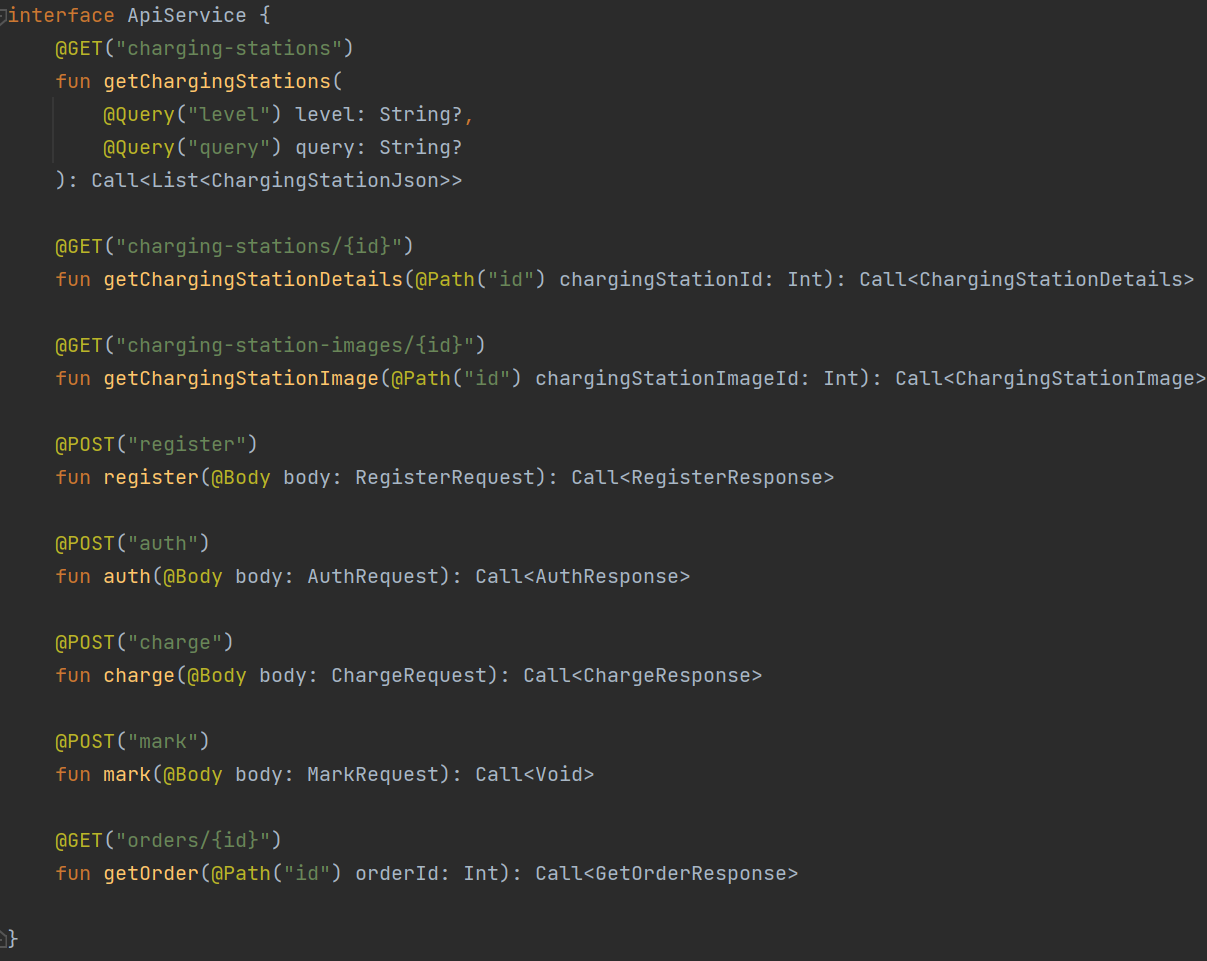


Рисунок 29 – интерфейс ApiService

Одним из ключевых примеров использования Retrofit в приложении стало получение данных о зарядных станциях. Метод getChargingStations с аннотацией @GET отправляет GET-запрос на сервер для получения списка зарядных станций. Он принимает параметры через аннотацию @Query, что позволяет фильтровать станции по уровню заряда или по поисковому запросу. Параметры уровня и запроса могут быть переданы как null, если они не нужны, что добавляет гибкости в работе с этой конечной точкой.

Кроме того, для получения детальной информации о конкретной зарядной станции был реализован метод getChargingStationDetails. В данном методе используется аннотация @Path, позволяющая передавать идентификатор зарядной станции в URL. Это позволяет легко обращаться к конкретной записи на сервере и получать полные данные о выбранной зарядной станции.

Для работы с изображениями зарядных станций используется метод getChargingStationImage, который аналогично принимает идентификатор изображения через аннотацию @Path, а сервер возвращает соответствующее изображение. Такой подход позволяет динамически загружать изображения на основе действий пользователя.

Помимо GET-запросов, в приложении реализованы и POST-запросы. Например, метод register используется для регистрации новых пользователей. В теле запроса отправляется объект с данными регистрации, который автоматически сериализуется в JSON и передается на сервер. Аналогично, метод auth выполняет аутентификацию пользователя, отправляя данные для входа (адрес электронной почты и пароль).

Запросы, связанные с процессом зарядки, были реализованы через методы charge и mark. Оба метода отправляют POST-запросы на сервер, передавая информацию в теле запроса через аннотацию @Body. Это позволяет передавать объекты с необходимыми данными для обработки процесса зарядки или отметки зарядной станции.

Для получения информации о заказах был реализован метод getOrder, который использует аннотацию @GET и @Path, чтобы по идентификатору заказа возвращать данные о конкретном заказе.

Использование Retrofit в приложении значительно упростило процесс работы с сетью. Преимущество Retrofit заключается в том, что он абстрагирует многие сложные детали, связанные с выполнением HTTP-запросов, такими как создание, обработка ошибок, сериализация и десериализация данных. В результате код стал более чистым, структурированным и легко поддерживаемым.

После установки приложения пользователь увидит экран приветствия SplashActivity, который загрузится первым. Этот экран отображается на короткое время и выполняет начальную инициализацию приложения. Если это первый запуск приложения, то после SplashActivity пользователю будет показан экран AuthenticationActivity. На этом экране пользователь может войти в свою учетную запись или зарегистрироваться. Также предусмотрена кнопка «Пропустить», которая позволяет перейти к основному функционалу приложения без регистрации или авторизации, если это предпочтительно.

После авторизации или пропуска экрана данного этапа пользователь будет перенаправлен на MainActivity, который является основным экраном приложения. В MainActivity пользователю доступны три основных способа выбора зарядной станции:

1. Поиск на карте: пользователь видит карту, на которой отображаются все доступные зарядные станции. Это позволяет визуально искать и выбирать ближайшие станции, просматривая их расположение на карте.

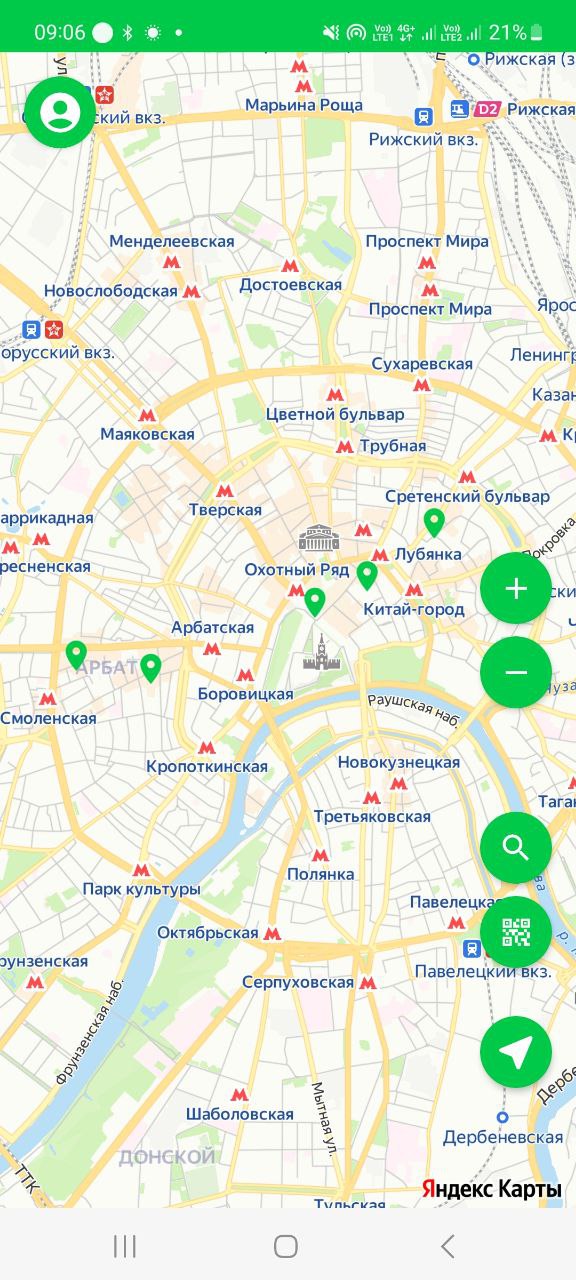


Рисунок 30 – главный экран (карта)

2. Поиск по списку: альтернативный способ позволяет пользователю воспользоваться функцией поиска по списку зарядных станций. Это удобный способ, если пользователь предпочитает искать станции по названиям, адресам или другим критериям.

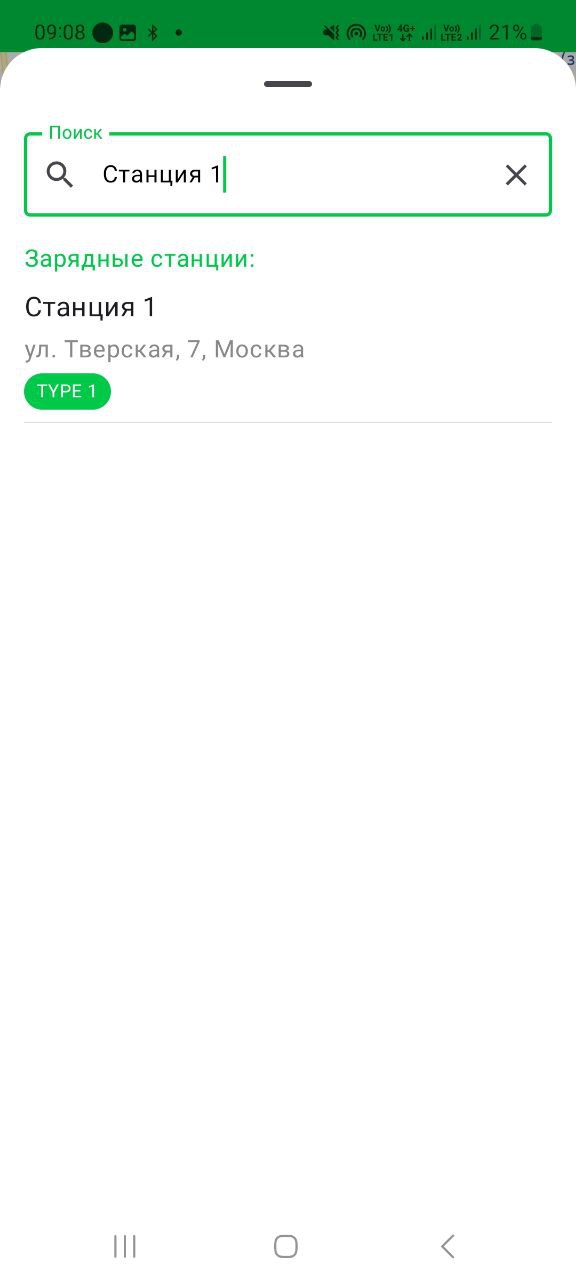


Рисунок 31 – главный экран (список)

3. Сканирование QR-кода: для удобства пользователей также предусмотрена функция сканирования QR-кодов. Нажав на кнопку для сканирования, пользователь может использовать камеру устройства для сканирования QR-кодов на зарядных станциях и быстрого получения информации о выбранной станции.

После выбора зарядной станции открывается подробная информация о ней. На этом экране пользователю предоставляется возможность выбрать подходящий коннектор для зарядки.



Рисунок 32 – главный экран (информация о зарядной станции)

Кроме того, на главном экране MainActivity предусмотрена кнопка для входа в учетную запись. Нажав на эту кнопку, пользователь может войти в свой аккаунт или зарегистрироваться, если это еще не было сделано ранее.

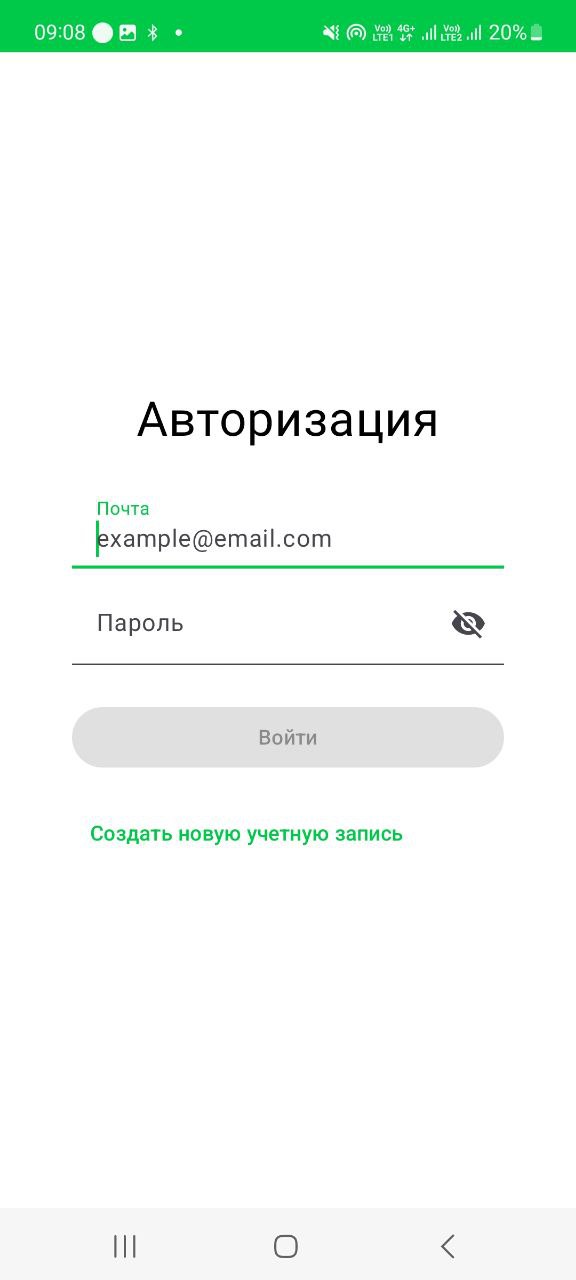


Рисунок 33 – экран входа в учетную запись

## Тестирование

В рамках тестирования приложения было проведено несколько ключевых проверок, чтобы убедиться в корректности работы всех функциональных элементов и их интеграции с базой данных и сервером.

Для начала проверим, что приложение корректно отображает данные, содержащиеся в базе данных. После запуска приложения и перехода к экрану с картой, мы видим 5 меток, представляющих зарядные станции. Это подтверждается изображением, представленным на рисунке 34.

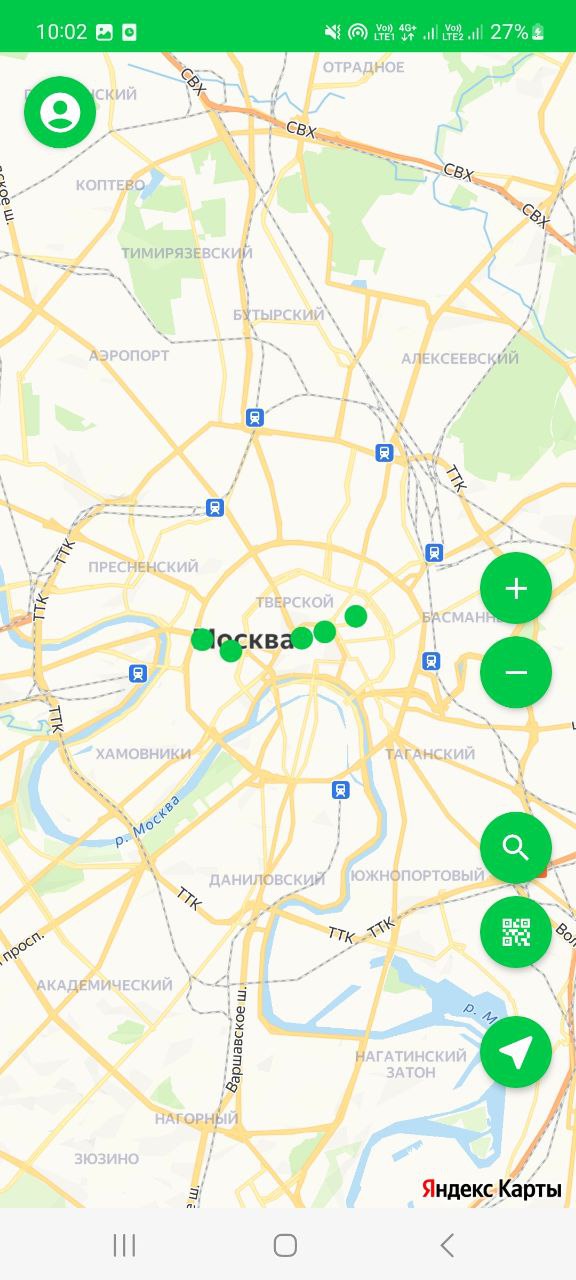


Рисунок 34 – карта с текущими зарядными станциями

Мы также проверяем, что запрос на получение списка зарядных станций отработал корректно и данные, отображаемые на карте, соответствуют записям в базе данных. Этот этап тестирования позволяет убедиться в том, что информация, представляемая пользователю, актуальна и верно отражает данные, хранящиеся в системе.

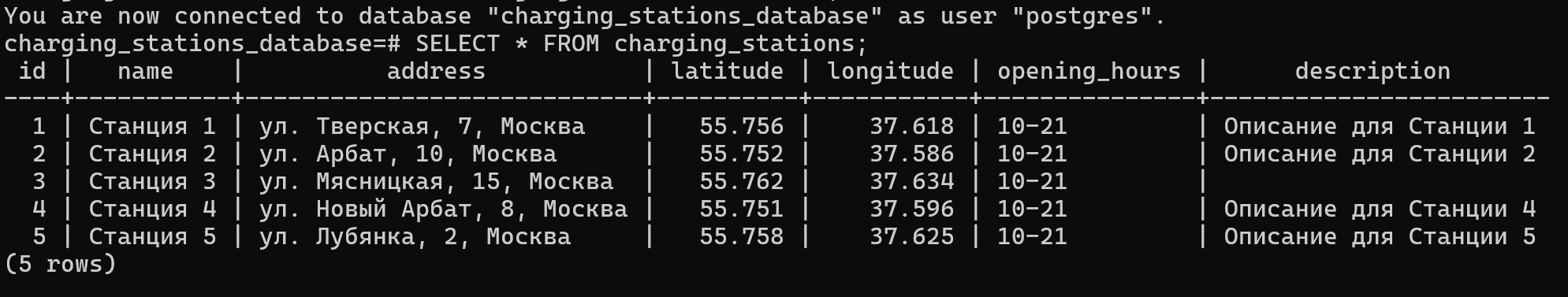


Рисунок 35 – данные в таблице charging\_stations

Открываем станции на карте и убеждаемся, что все данные действительно совпадают.



Рисунок 36 – информация по зарядной станции 1

Далее, для проверки корректности работы удаления данных, мы попробуем удалить зарядную станцию 1 с сервера.

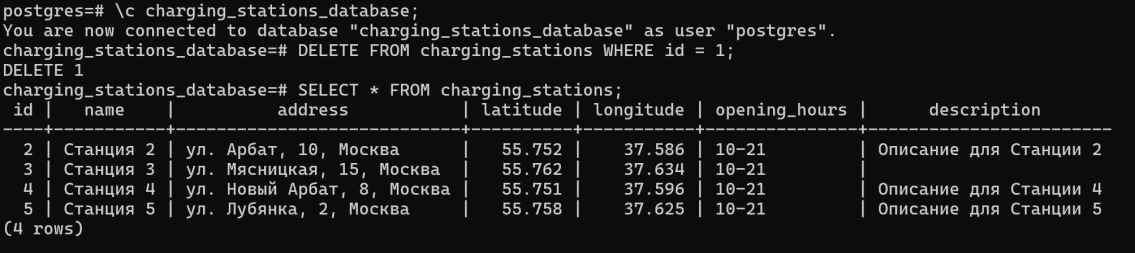


Рисунок 37 – удаление зарядной станции 1

После выполнения этого действия мы наблюдаем результат на экране приложения, который должен отразить изменения.

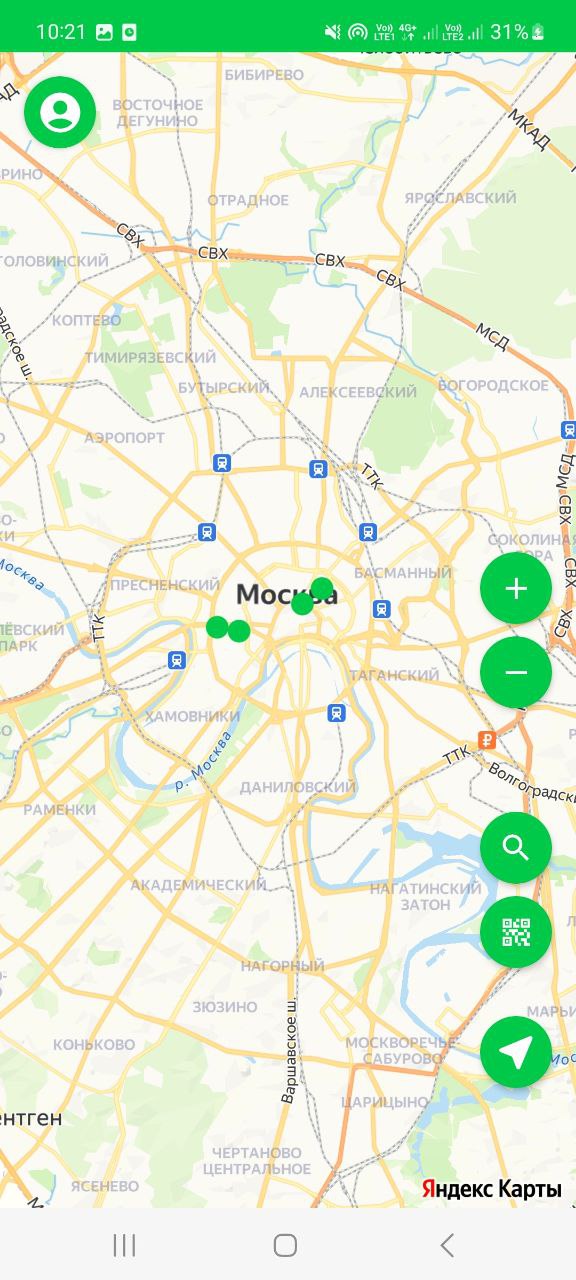


Рисунок 38 – карта зарядных станций после изменений на сервере

Также мы проверяем, что коннекторы удаляются каскадно, то есть одновременно с удалением станции, все связанные записи о коннекторах также исчезают из базы данных.

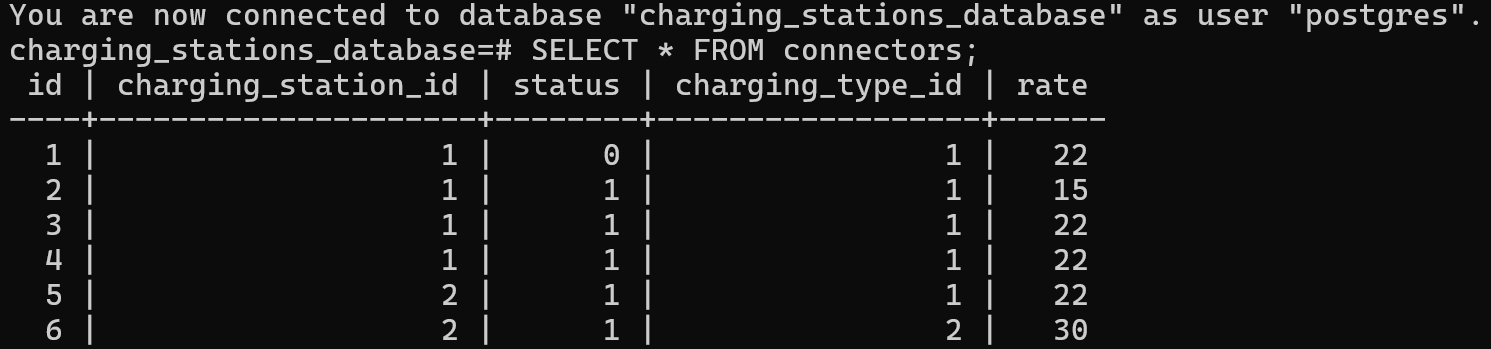


Рисунок 39 – таблица connectors до удаления станции

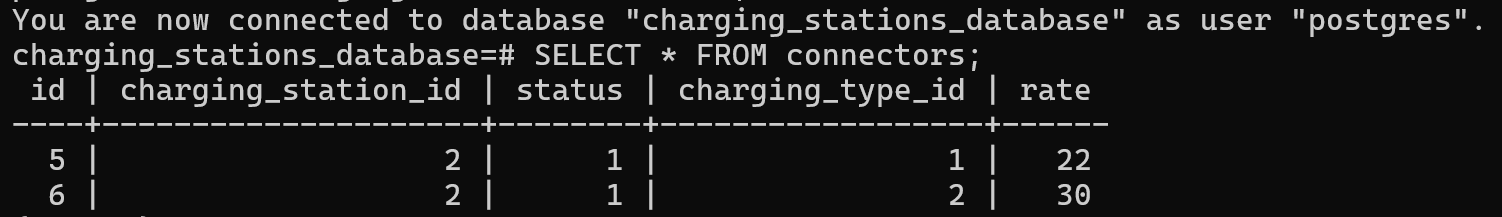


Рисунок 40 – таблица connectors после удаления станции

Кроме того, убедимся, что мы не можем удалять тип зарядки, который используется, например, тип зарядки с идентификатором 1.

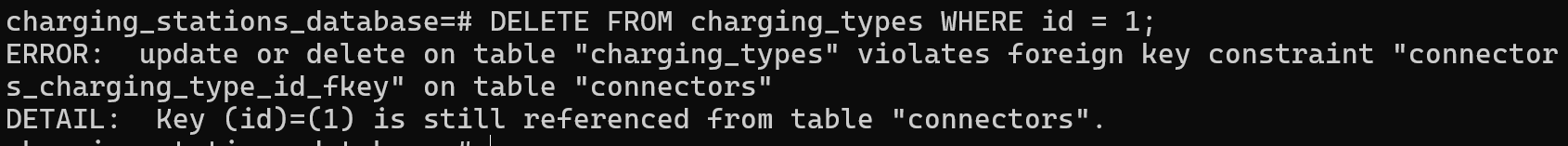


Рисунок 41 – результат попытки удаления используемого типа зарядки

Удостоверимся, что изменение внешнего ключа charging\_station\_id запрещено.

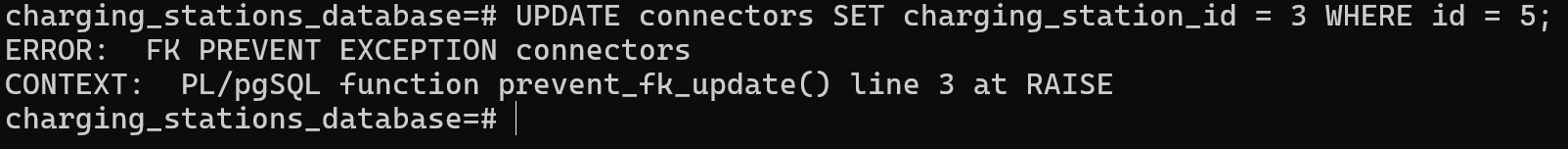


Рисунок 42 – результат попытки изменения внешнего ключа charging\_station\_id.

Одним из ключевых этапов тестирования стала проверка валидации полей ввода. Формы регистрации, авторизации, а также другие экраны, где требуется ввод данных пользователем, обрабатывают информацию корректно. Было проверено, что при попытке отправить некорректные данные (например, пустые поля, неверные форматы электронной почты или пароля) пользователю отображаются соответствующие сообщения об ошибках, предотвращая дальнейшие действия до исправления введенной информации. Это важный аспект, обеспечивающий защиту данных и предотвращение ошибок на уровне пользовательского ввода.

Также были протестированы другие элементы интерфейса и функциональные возможности приложения. Проверялись сценарии переходов между экранами, корректность работы кнопок и интуитивное поведение интерфейса. Пользовательские действия обрабатываются корректно, а система стабильно реагирует на нажатия и другие интерактивные элементы. Особое внимание уделялось проверке работы карты и всех функций, связанных с выбором зарядных станций: через поиск, QR-код и непосредственно на карте.

Были также протестированы сценарии с сетевыми ошибками, такими как отсутствие интернет-соединения или проблемы с сервером. Было проверено, что в таких ситуациях приложение корректно уведомляет пользователя о проблемах, предоставляет возможность повторить действие или дождаться восстановления соединения. Это обеспечивает улучшенное взаимодействие с пользователем, даже в условиях нестабильной сети.

# ЗАКЛЮЧЕНИЕ

В ходе данной курсовой работы были успешно реализованы ключевые компоненты мобильного приложения для зарядных станций электромобилей, включая серверную часть, базу данных и API, обеспечивающие обработку и хранение данных, а также мобильное приложение, предоставляющее удобный пользовательский интерфейс. Основной целью было создание приложения, способного эффективно управлять процессом поиска, бронирования и использования зарядных станций.

Проектирование и разработка серверной части обеспечила обработку запросов от мобильного клиента и взаимодействие с базой данных. Также был разработан API для взаимодействия мобильного приложения с сервером, что позволило передавать данные о зарядных станциях в реальном времени и поддерживать актуальность информации.

Одним из ключевых элементов проекта стала разработка базы данных, которая обеспечила хранение информации о пользователях, зарядных станциях и заказах. База данных была спроектирована таким образом, чтобы поддерживать целостность данных и их быструю синхронизацию между сервером и мобильным приложением. Это позволило пользователям получать актуальные данные о доступности зарядных станций и бронировать их в реальном времени.

Особое внимание было уделено разработке мобильного приложения. Приложение предоставило пользователям простой и интуитивно понятный интерфейс, позволяющий легко отслеживать их загруженность и осуществлять бронирование. Все функциональные элементы были протестированы, что подтвердило их корректную работу и удобство использования.

Таким образом, курсовая работа продемонстрировала важность комплексного подхода к разработке мобильных приложений, включающего серверную часть, базу данных, API и клиентскую часть. В процессе работы были решены задачи синхронизации данных, проектирования удобного интерфейса и обеспечения надежного взаимодействия всех компонентов системы.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Интернет-ресурс: https://developer.android.com
2. Основы технологий баз данных: учеб. пособие / Б. А. Новиков, Е. А. Горшкова, Н. Г. Графеева; под ред. Е. В. Рогова. — 2-е изд. — М.: ДМК Пресс, 2020. — 582 с.
3. Clean Architecture: A Craftsman’s Guide to Software Structure and Design, Robert C. Martin, 2017
4. Kotlin and Android Development featuring Jetpack, Michael Fazio, 2021
5. HTTP: The Definitive Guide, David Gourley, Brian Totty, 2002