Room Хранение данных на Android для всех и каждого

<https://www.bignerdranch.com/blog/room-data-storage-for-everyone/>

* [Java](https://habr.com/hub/java/),
* [Разработка мобильных приложений](https://habr.com/hub/mobile_dev/),
* [Разработка под Android](https://habr.com/hub/android_dev/)
* Перевод
* Tutorial

**Room** — это новый способ сохранить данные приложений в Android-приложении, представленный в этом году на *Google I/O*. Это часть новой*Android Architecture*, группа библиотек от Google, которые поддерживают уместную архитектуру приложений. Room предлагается в качестве альтернативы Realm, ORMLite, GreenDao и многим другим.

**Room** — это высокоуровневый интерфейс для низкоуровневых привязок SQLite, встроенных в Android, о которых вы можете узнать больше в [документации](https://developer.android.com/training/basics/data-storage/databases.html). Он выполняет большую часть своей работы во время компиляции, создавая API-интерфейс поверх встроенного SQLite API, поэтому вам не нужно работать с *Cursor* или *ContentResolver*.

Использование Room

Во-первых, [добавьте](https://developer.android.com/topic/libraries/architecture/adding-components.html) Room в свой проект. После этого вам нужно будет передать в Room, как выглядят ваши данные. Предположим, имеется простой класс модели, который выглядит следующим образом:

public class Person {

String name;

int age;

String favoriteColor;

}

Чтобы рассказать **Room** о классе **Person**, добавляем аннотицию [*Entity*](https://habrahabr.ru/users/entity/) к классу и *@PrimaryKey* к ключу:

@Entity

public class Person {

@PrimaryKey String name;

int age;

String favoriteColor;

}

Благодаря этим двум аннотациям **Room** теперь знает, как создать таблицу для хранения экземпляров *Person*.

Важная вещь, которую следует учитывать при настройке ваших моделей: каждое поле, которое хранится в базе данных, должно быть общедоступным или иметь геттер и сеттер в стандартном стиле **Java Beans** (например, *getName ()* и *setName (имя строки)*).

В классе *Person* теперь есть вся информация, которая требуется **Room** для создания таблиц, но у вас нет способа фактически добавлять, запрашивать или удалять данные из базы данных. Вот почему вам нужно будет сделать объект доступа к данным (DAO). **DAO** предоставляет интерфейс в самой базе данных и занимается манипулированием хранимыми данными Person.

Вот простой интерфейс DAO для класса Person:

@Dao

public interface PersonDao {

*// Добавление Person в бд*

@Insert

void insertAll(Person... people);

*// Удаление Person из бд*

@Delete

void delete(Person person);

*// Получение всех Person из бд*

@Query("SELECT \* FROM person")

List<Person> getAllPeople();

*// Получение всех Person из бд с условием*

@Query("SELECT \* FROM person WHERE favoriteColor LIKE :color")

List<Person> getAllPeopleWithFavoriteColor(String color);

}

Первое, что нужно заметить, это то, что **PersonDao** — это интерфейс, а не класс. Другая интересная деталь — это инструкции SQL в аннотациях *[Query](https://habrahabr.ru/users/query/) ()*. Операторы SQL говорят Room, какую информацию вы хотите получить из базы данных. Они также проверяются во время компиляции. Поэтому, если вы измените подпись метода List getAllPeopleWithFavoriteColor ( **название цвета** ) на List getAllPeopleWithFavoriteColor ( **int color** ), Room выдаст ошибку во время компиляции:

incompatible types: int cannot be converted to String

И если вы сделаете опечатку в выражении SQL, например, напишите *favoriteColors (* **множественное число** *)* вместо *favoriteColor (* **единственное число** *)*, Room также выдаст ошибку компиляции:

There is a problem with the query: [SQLITE\_ERROR] SQL error or missing database (no such column: favoriteColors)

Вы не можете получить экземпляр PersonDao, потому что это интерфейс. Чтобы иметь возможность использовать классы DAO, вам необходимо создать класс базы данных. За кулисами этот класс будет отвечать за ведение самой базы данных и предоставление экземпляров DAO.

Вы можете создать свой класс базы данных всего за пару строк:

@Database(entities = {Person.class */\*, AnotherEntityType.class, AThirdEntityType.class \*/*}, version = 1)

public abstract class AppDatabase extends RoomDatabase {

public abstract PersonDao getPersonDao();

}

Это лишь описание структуры базы данных, но сама база данных будет жить в одном файле. Чтобы получить экземпляр **AppDatabase**, сохраненный в файле с именем **populus-database**, вы должны написать:

AppDatabase db = Room.databaseBuilder(getApplicationContext(),

AppDatabase.class, "populus-database").build();

Если вы хотите получить все данные обо всех Person, которые находятся в базе данных, вы могли бы написать:

List<Person> everyone = db.getPersonDao().getAllPeople();

Преимущества использования Room

В отличие от большинства ORM, **Room** использует обработчик аннотации для выполнения всей своей манеры сохранения данных. Это означает, что ни ваши классы приложений, ни классы моделей не должны ничего расширять в **Room**, в отличие от многих других ORM, включая Realm и SugarORM. Как вы видели при ошибках с аннотациями [Query](https://habrahabr.ru/users/query/) () выше, вы также получаете возможность проверки корректности SQL-запросов во время компиляции, что может сэкономить вам много хлопот.

Room также позволяет вам наблюдать за изменениями данных, интегрируя их как с [API LiveData](https://developer.android.com/topic/libraries/architecture/livedata.html)Архитектурных Компонентов, так и с RxJava 2. Это означает, что если у вас сложная схема, где изменения в базе данных должны появляться в нескольких местах вашего приложения, Room делает уведомления об изменениях. Это мощное дополнение может быть включено одной строкой. Все, что вам нужно сделать, это изменить тип возвращаемых значений.

Например, этот метод:

@Query("SELECT \* FROM person")

List<Person> getAllPeople();

Становится следующим:

@Query("SELECT \* FROM person")

LiveData<List<Person>> */\* or Observable<List<Person>> \*/* getAllPeople();

Самое большое ограничение в Room: взаимосвязи

Самым большим ограничением в Room является то, что он не будет обрабатывать отношения с другими типами сущностей для вас автоматически, как и другие ORM. Это означает, что если вы хотите отслеживать домашних животных:

@Entity

public class Person {

@PrimaryKey String name;

int age;

String favoriteColor;

List<Pet> pets;

}

@Entity

public class Pet {

@PrimaryKey String name;

String breed;

}

То Room выдаст ошибку компиляци, так как не знает, как сохранить отношения между Person и Pet:

Cannot figure out how to save this field into database. You can consider adding a type converter for it.

Ошибка при компиляции предлагает конвертер типов, который преобразует объекты в примитивы, которые могут быть непосредственно сохранены в SQL. Поскольку List нельзя свести к примитиву, вам нужно сделать что-то другое. Это отношения «один ко многим», где у одного Person может быть много Pet. Room не может моделировать такие отношения, но она может справиться с обратными отношениями — у каждого Pet есть один Person. Чтобы смоделировать это, удалите поле для *Pet* в **Person** и добавьте поле *ownerId* в класс **Pet**:

@Entity

public class Person {

@PrimaryKey String name;

int age;

String favoriteColor;

}

@Entity(foreignKeys = @ForeignKey(

entity = Person.class,

parentColumns = "name",

childColumns = "ownerId"))

public class Pet {

@PrimaryKey String name;

String breed;

String ownerId; *// this ID points to a Person*

}

Это приведет к тому, что Room обеспечит ограничение внешнего ключа между объектами. Room не будет вызывать отношения «один-ко-многим» и «много-к-одному», но она дает вам инструменты для выражения этих отношений.

Чтобы получить всех домашних животных, принадлежащих конкретному человеку, вы можете использовать запрос, который находит всех домашних животных с данным идентификатором владельца. Например, вы можете добавить в свой DAO следующий метод:

@Query("SELECT \* FROM pet WHERE ownderId IS :ownerId")

List<Pet> getPetsForOwner(String ownerId);

Стоит ли использовать Room?

Если вы уже настроили сохранение данных в своем приложении и довольны им, то ничего не изменяйте. Каждая ORM и встроенная реализация SQLite будут продолжать работать так же, как и раньше. Room — это всего лишь еще один вариант сохранения данных.

Если вы используете SQLite или собираетесь использовать его, вы должны попробовать Room. Он обладает всеми возможностями, необходимыми для выполнения расширенных запросов, одновременно устраняя необходимость писать SQL-запросы для поддержки базы данных самостоятельно.