[Урок 6 Room Entity](https://startandroid.ru/ru/blog/27-course/architecture-components/530-urok-6-room-entity.html" \o "Урок 6. Room. Entity)

В этом уроке более подробно рассмотрим возможности Entity. Как задать имя таблицы. Как задать имя или тип поля. Как создать составной или внешний ключ. Как создать индекс. Как использовать вложенные объекты.

Имя таблицы

Entity класс используется для создания таблицы. По умолчанию в качестве имени таблицы используется имя этого класса. Но мы можем указать свое имя, используя параметр tableName.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Entity(tableName = "employees")  public class Employee {       // ...    } |

Для хранения объектов Employee будет создана таблица с именем employees.

Имя поля

По умолчанию в качестве имени полей в таблице используются имена полей Entity класса. Но мы можем указать свое имя, используя параметр name в аннотации [ColumnInfo](https://developer.android.com/reference/android/arch/persistence/room/ColumnInfo.html" \t "_blank).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Entity()  public class Employee {       @PrimaryKey()     public long id;       @ColumnInfo(name = "full\_name")     public String fullName;       public int salary;    } |

Для fullName в таблице будет создано поле с именем full\_name.

Тип поля

По умолчанию Room определяет тип данных для поля в таблице по типу данных поля в Entity классе. Но мы можем явно указать свой тип.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Entity()  public class Employee {       @PrimaryKey(autoGenerate = true)     public long id;       public String name;       @ColumnInfo(typeAffinity = TEXT)     public int salary;    } |

В таблице поле salary будет с типом TEXT.

Модификаторы доступа

Чтобы Room мог добраться до полей класса Entity, мы делаем их public.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Entity  public class Employee {       @PrimaryKey     public long id;       public String name;       public int salary;    } |

Но есть возможность использовать private поля. Для этого надо добавить set/get методы.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | @Entity  public class Employee {       @PrimaryKey     private long id;       private String name;       private int salary;       public long getId() {         return id;     }       public void setId(long id) {         this.id = id;     }       public String getName() {         return name;     }       public void setName(String name) {         this.name = name;     }       public int getSalary() {         return salary;     }       public void setSalary(int salary) {         this.salary = salary;     }    } |

Все поля - private. Но каждое имеет set/get методы.

В Android Studio эти методы добавляются парой кликов. Жмете в коде ALT+INSERT, выбираете пункт Getter and Setter, затем выбираете поля, для которых надо создать методы.

Вместо set-методов мы также можем использовать конструктор.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @Entity  public class Employee {       public Employee(long id) {         this.id = id;     }       @PrimaryKey     private long id;       public String name;       public int salary;         public long getId() {         return id;     }    } |

Поле id здесь - private и имеет get-метод. А вместо set-метода, Room будет использовать конструктор.

Параметр конструктора должен иметь тот же тип и имя, что и поле Entity класса. Вы можете использовать конструктор для всех полей или только для некоторых, как в примере выше.

Я для упрощения примеров везде буду использовать public поля.

Первичный ключ

Мы уже знаем, как с помощью @PrimaryKey назначить какое-либо поле ключом. Каждый Entity класс должен содержать хотя бы одно такое поле. Даже если в классе всего одно поле.

У PrimaryKey есть параметр autoGenerate. Он позволяет включить для поля режим autoincrement, в котором база данных сама будет генерировать значение, если вы его не укажете.

|  |  |
| --- | --- |
| 1  2 | @PrimaryKey(autoGenerate = true)  public long id; |

Теперь при создании Entity объекта вы можете не заполнять поле id. База сама найдет ближайшее свободное значение и использует его.

Чтобы создать составной ключ, используйте параметр primaryKeys.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Entity(primaryKeys = {"key1", "key2"})  public class Item {       public long key1;     public long key2;       // ...    } |

Внешний ключ

Внешние ключи позволяют связывать таблицы между собой. Если вы еще не знакомы с ними, то можете почитать о них в инете.

В выше рассмотренных примерах у нас есть класс Employee для хранения данных по сотрудникам. Давайте создадим класс Car для хранения данных по машинам. И каждая машина должна быть прикреплена к какому-либо сотруднику.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Entity(foreignKeys = @ForeignKey(entity = Employee.class, parentColumns = "id", childColumns = "employee\_id"))  public class Car {       @PrimaryKey(autoGenerate = true)     public long id;       public String model;       public int year;       @ColumnInfo(name = "employee\_id")     public long employeeId;    } |

В поле employee\_id будет храниться id сотрудника, к которому прикреплена эта машина.

Используем параметр foreignKeys для создания внешнего ключа. Указываем, что значения поля employee\_id (параметр childColumns) должно обязательно быть равно какому-либо значению поля id (параметр parentColumns) в таблице сотрудников Employee (параметр entity).

Т.е. если у нас есть три сотрудника с id 1,2 и 3, мы не сможем добавить в базу данных машину с employee\_id = 4. Потому что в базе нет такого родительского ключа, т.е. сотрудника с id = 4.

Или, если вы попытаетесь удалить родительский ключ, т.е. сотрудника, к которому прикреплена какая-либо машина, то база выдаст вам ошибку. Потому что после удаления сотрудника, у машины в поле employee\_id будет находиться значение, которого нет в поле id таблицы сотрудников.

Для подобных случаев удаления или изменения родительского ключа, вы можете настроить поведение базы данных. По умолчанию она возвращает ошибку, но это можно поменять с помощью параметров onDelete и onUpdate в аннотации ForeignKey.

Добавим параметр onDelete

|  |  |
| --- | --- |
| 1 | @Entity(foreignKeys = @ForeignKey(entity = Employee.class, parentColumns = "id", childColumns = "employee\_id", onDelete = CASCADE)) |

Его значение = CASCADE. Это означает, что при удалении родительского ключа, будут удалены, связанные с ним дочерние ключи. Т.е. при удалении сотрудника, удалится и его машина.

Список возможных значений для параметра onDelete можно посмотреть [здесь](https://developer.android.com/reference/android/arch/persistence/room/ForeignKey.html). А подробнее почитать о них на русском [здесь](http://xbb.uz/SQLite/Poddjerzhka-vnjeshnih-kljuchjej-v-SQLite/4.3.html).

Еще один параметр аннотации ForeignKey - это deferred, имеющий по умолчанию значение false. Если задать этому параметру значение true, то внешний ключ станет отложенным. Это может быть полезно при вставке данных в разные таблицы в рамках одной транзакции. Вы сможете внести все необходимые изменения, и проверка на корректность внешних ключей будет выполнена в самом конце, при выполнении commit.

Подробнее об этом можно почитать [здесь](http://xbb.uz/SQLite/Poddjerzhka-vnjeshnih-kljuchjej-v-SQLite/4.html#fk_deferred).

Индекс

Индексы могут повысить производительность вашей таблицы. Если вы еще не знакомы с ними, то можете почитать о них в инете.

В аннотации Entity есть параметр indicies, который позволяет задавать индексы.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | @Entity(indices = {                 @Index("salary"),                 @Index(value = {"first\_name", "last\_name"})             }         )  public class Employee {       @PrimaryKey(autoGenerate = true)     public long id;       @ColumnInfo(name = "first\_name")     public String firstName;       @ColumnInfo(name = "last\_name")     public String lastName;       public int salary;    } |

Создаем два индекса: один по полю salary, а другой по двум полям first\_name и last\_name.

Индекс дает возможность установить для его полей проверку на уникальность. Это делается параметром unique = true.

|  |  |
| --- | --- |
| 1 | @Entity(indices = {@Index(value = {"first\_name", "last\_name"}, unique = true)}) |

В этом случае база будет следить, чтобы в этой таблице не было записи с повторящейся парой значений first\_name и last\_name.

Индекс для одного поля также может быть настроен через параметр index аннотации ColumnInfo

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Entity()  public class Employee {       @PrimaryKey(autoGenerate = true)     public long id;       public String name;       @ColumnInfo(index = true)     public int salary;    } |

Будет создан индекс для поле salary.

Вложенные объекты

Пусть у нас есть класс Address, с данными о адресе. Это обычный класс, не Entity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Address {       public String city;     public String street;     public int number;    } |

И мы хотим использовать его в Entity классе Employee

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Entity()  public class Employee {       @PrimaryKey(autoGenerate = true)     public long id;       public String name;       public int salary;       public Address address;    } |

Если мы сделаем так, то Room будет ругаться, т.к. он не знает, как сохранить такой объект в базу:   
*Cannot figure out how to save this field into database. You can consider adding a type converter for it.*

Но есть простое решение - использовать аннотацию [Embedded](https://developer.android.com/reference/android/arch/persistence/room/Embedded.html).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Entity()  public class Employee {       @PrimaryKey(autoGenerate = true)     public long id;       public String name;       public int salary;       @Embedded     public Address address;    } |

Embedded подскажет Room, что надо просто взять поля из Address и считать их полями таблицы Employee.

Т.е. в базе будет создана таблица Employee с полями id, name, salary, city, street, number.

Добавление новой записи будет выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Employee employee = new Employee();  employee.id = 1;  employee.name = "John Smith";  employee.salary = 10000;  Address address = new Address();  address.city = "London";  address.street = "Baker Street";  address.number = 221;  employee.address = address;    db.employeeDao().insert(employee); |

Мы создаем вложенный объект Address, но Room разберется, и запишет все в таблицу, как плоскую структуру.

Embedded объекты могут включать в себя другие Embedded объекты.

Если у вас получается так, что совпадают имена каких-то полей в основном объекте и в Embedded объекте, то используйте префикс для Embedded объекта.

|  |  |
| --- | --- |
| 1  2 | @Embedded(prefix = "address")  public Address address; |

В этом случае к именам полей Embedded объекта в таблице будет добавлен указанный префикс.

Ignore

Аннотация [Ignore](https://developer.android.com/reference/android/arch/persistence/room/Ignore.html" \t "_blank) позволяет подсказать Room, что это поле не должно записываться в базу или читаться из нее.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Entity  public class Employee {       @PrimaryKey     public long id;       public String name;       public int salary;       @Ignore     public Bitmap avatar;    } |

Нам не нужно хранить Bitmap в базе, поэтому добавляем Ignore к этому полю.