FIT2081 Mobile application development - S1 2023 MUM

Database@Android

This week, we will learn how to add a local database to Android applications.

What is a database?

A database is an organized collection of **structured** information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). [oracle.com]
Android uses SQLite as a database management system.

What language does SQLite use to communicate?

SQLite uses SQL language (Structured Query Language)

So, what is SQL?

SQL is a language that allows you to access and manipulate databases. SQL statements are used to execute tasks such as adding data to a database or retrieving data from a database. SQL is used by nearly all relational databases to query, manipulate, and define data and provide access control. Now, let's have an example. The depicted below is an excerpt from a table called 'Customers' [Ref WeSchools] and you can notice the following:

- Tables consist of rows and columns
- Each column has a name
- Each row has the same set of columns
- Each cell stores one value only
- The data in a column have the same data type
- The table has a column that works as an index (CustomerID). This column is unique and cannot contain NULL values. This column is called the 'primary key'.

CustomerID	CustomerName	ContactName	Address	City
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå
6	Blauer See Delikatessen	Hanna Moos	Forsterstr. 57	Mannheim
7	Blondel père et fils	Frédérique Citeaux	24, place Kléber	Strasbourg
8	Bólido Comidas preparadas	Martín Sommer	C/ Araquil, 67	Madrid
9	Bon app'	Laurence	12, rue des	Marseille

		Lebihans	Bouchers	
10	Bottom-Dollar Marketse	Elizabeth Lincoln	23 Tsawassen Blvd.	Tsawassen

What if we need to retrieve the customers who live in Germany, then we would call:

```
SELECT * FROM Customers WHERE Country='Germany';
```

Where:

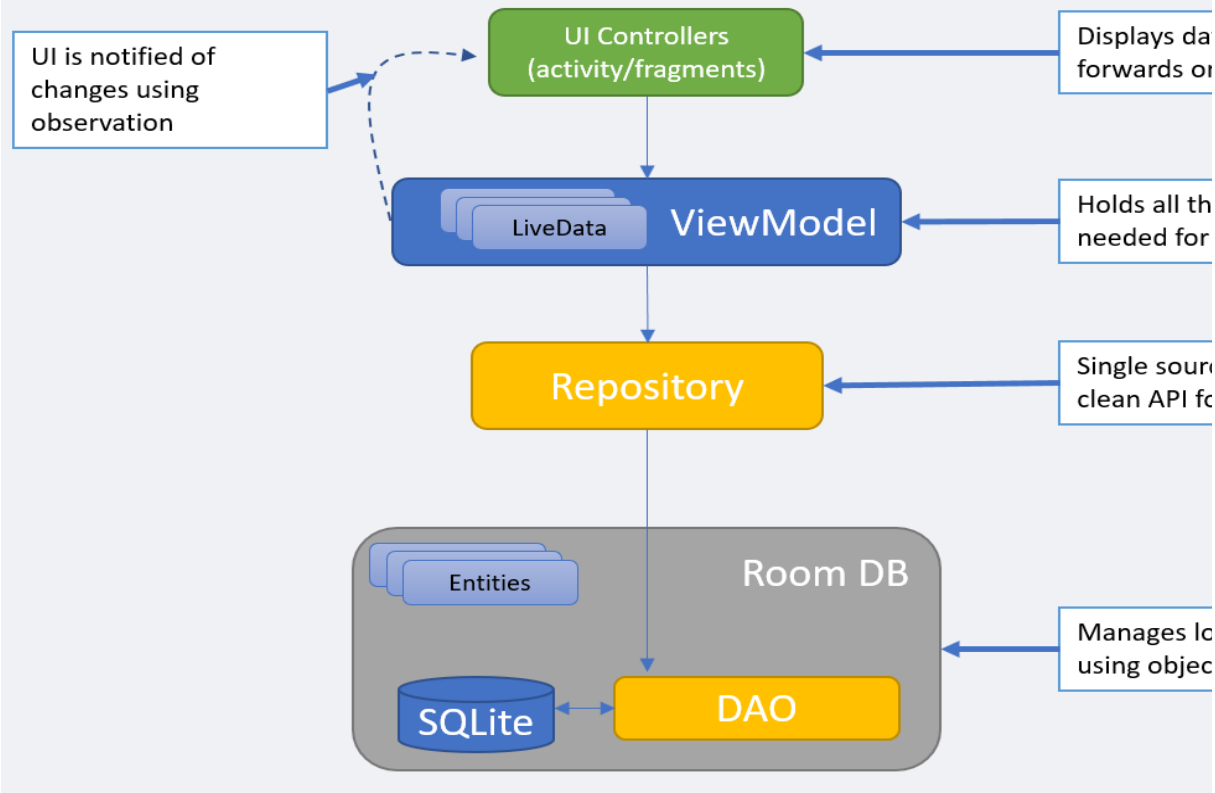
- 'Customers' is the table name
- '*' means fetch all the columns
- 'WHERE' is the row filter. It selects the row that matches the provided criteria.

Number of Records: 11

CustomerID	CustomerName	ContactName	Address	City
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin
6	Blauer See Delikatessen	Hanna Moos	Forsterstr. 57	Mannheim
17	Drachenblut Delikatessend	Sven Ottlieb	Walserweg 21	Aachen
25	Frankenversand	Peter Franken	Berliner Platz 43	München
39	Königlich Essen	Philip Cramer	Maubelstr. 90	Brandenburg
44	Lehmanns Marktstand	Renate Messner	Magazinweg 7	Frankfurt a.M.
52	Morgenstern	Alexander Feuer	Heerstr. 22	Leipzig

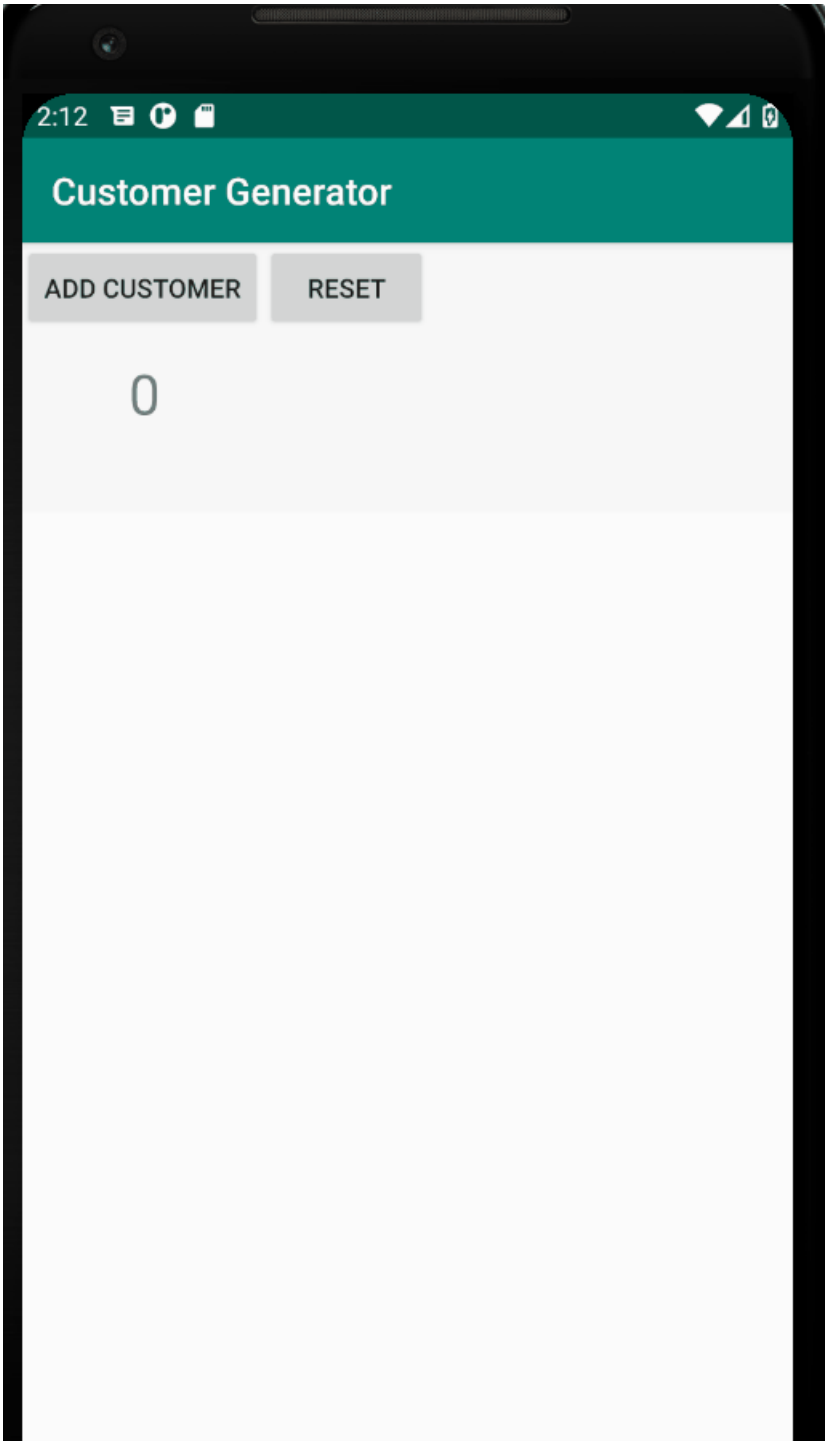
As you can see, all the values in the last column 'Country' are Germany.

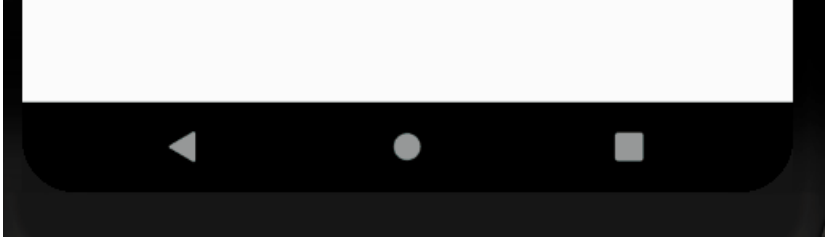
SQLite, Rooms, and LiveData



- *SQLite database: is a relational database management system that is used by Android to store relational data.*
- *DAO: Data Access Objects are the main classes where you define your database interactions. They can include a variety of query methods.*
- Entities: Each entity represents one table in the database
- *Room database:* The room database object provides the interface to the underlying SQLite database.
- *Repository:* It's a class that contains all of the code necessary for directly handling all data sources used by the application. This avoids the need for the UI controller and ViewModel to contain code that directly accesses sources such as databases or web services.
- ViewModel: A ViewModel object provides the data for a specific UI component, such as a fragment or activity, and contains data-handling business logic to communicate with the model.
- LiveData: It's a data holder that allows a value to become observable. In other words, an observable object can notify other objects when changes to its data occur, thereby solving the problem of ensuring that the user interface always matches the data within the ViewModel.

Let's develop an application that generates customers randomly. Each customer is represented by a name and address as shown below:





Entities

As aforementioned, an entity is a Java class that defines a table in the database.

Now, let's create a table of the following attributes:

Name in Java	Name in DB	Datatype	Attributes
id	customerId	integer	Primary Key NonNull
name	customerName	string	
address	customerAddress	string	

```
1   package com.fit2081.rooms.provider;
2
3   import androidx.annotation.NonNull;
4   import androidx.room.ColumnInfo;
5   import androidx.room.Entity;
6   import androidx.room.PrimaryKey;
7
8   @Entity(tableName = "customers")
9   public class Customer {
10      @PrimaryKey(autoGenerate = true)
11      @NonNull
12      @ColumnInfo(name = "customerId")
13      private int id;
14      @ColumnInfo(name = "customerName")
15      private String name;
16      @ColumnInfo(name = "customerAddress")
17      private String address;
18
19      public Customer(String name, String address) {
20          this.name = name;
21          this.address = address;
22      }
23
24      public int getId() {
25          return id;
26      }
27
28      public String getName() {
29          return name;
30      }
31
32      public String getAddress() {
33          return address;
34      }
35
36      public void setId(@NonNull int id) {
```

```
37          this.id = id;
38      }
39
40  }
```

Notes

- line@8: the annotation @Entity is required to define the class as a Room Entity. It also specifies the table name.
- the class 'Customers' has three attributes id, name, and address. Each attribute has an annotation 'ColumnInfo' that specifies the column name in the database
- line@10: the annotation '@PrimaryKey' makes the id as a primary key for the current table which the ' @NonNull' ensures that this column will not be saved without a value.

Room Database

In this class, we will define the Room Database (database might contain one or more tables)

```
1   package com.fit2081.rooms.provider;
2   import android.content.Context;
3   import androidx.room.Database;
4   import androidx.room.Room;
5   import androidx.room.RoomDatabase;
6   import java.util.concurrent.ExecutorService;
7   import java.util.concurrent.Executors;
8
9
10  @Database(entities = {Customer.class}, version = 1)
11  public abstract class CustomerDatabase extends RoomDatabase {
12
13      public static final String CUSTOMER_DATABASE_NAME =
14  "customer_database";
15
16      public abstract CustomerDao customerDao();
17
18      // marking the instance as volatile to ensure atomic access to the
19  variable
20      private static volatile CustomerDatabase INSTANCE;
21      private static final int NUMBER_OF_THREADS = 4;
22      static final ExecutorService databaseWriteExecutor =
23              Executors.newFixedThreadPool(NUMBER_OF_THREADS);
24
25      static CustomerDatabase getDatabase(final Context context) {
26          if (INSTANCE == null) {
27              synchronized (CustomerDatabase.class) {
28                  if (INSTANCE == null) {
29                      INSTANCE =
30  Room.databaseBuilder(context.getApplicationContext(),
31                              CustomerDatabase.class, CUSTOMER_DATABASE_NAME)
32                              .build();
33                  }
34              }
35          }
36          return INSTANCE;
37      }
38
39  }
```

Note:

- Line@10: The '@Database' annotation is required to consider the current class as a Room database. It specifies the list of entities and the current version. The version is required for upgrading or downgrading the current scheme.
- Line@20: the 'databaseWriteExecutor' instance will be used by the repository to execute the DAO methods
- Line@23: the 'getDatabase' method returns a reference to the current database instance if it is not null. Otherwise, it creates a new instance using 'Room.databaseBuilder()', which needs as input the context, a reference to the Room Database class, and a name for the database.

DAO

The Dao is an interface that defines the database operations that should be performed on the database.

```
1   package com.fit2081.rooms.provider;
2
3   import androidx.lifecycle.LiveData;
4   import androidx.room.Dao;
5   import androidx.room.Insert;
6   import androidx.room.Query;
7
8   import java.util.List;
9
10  @Dao
11  public interface CustomerDao {
12
13      @Query("select * from customers")
14      LiveData<List<Customer>> getAllCustomer();
15
16      @Query("select * from customers where customerName=:name")
17      List<Customer> getCustomer(String name);
18
19      @Insert
20      void addCustomer(Customer customer);
21
22      @Query("delete from customers where customerName= :name")
23      void deleteCustomer(String name);
24
25      @Query("delete FROM customers")
26      void deleteAllCustomers();
27  }
```

Notes:

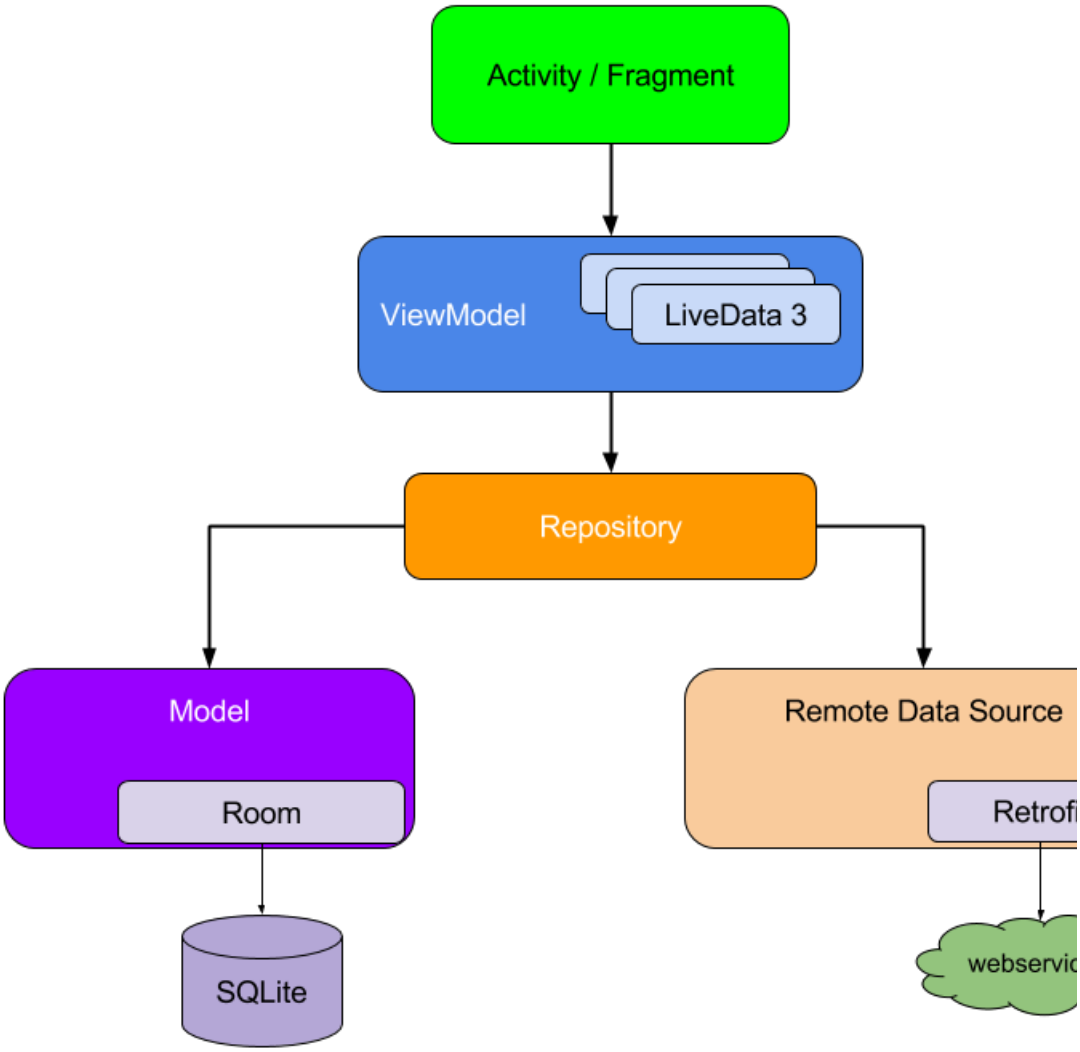
- Line@10: the annotation '@Dao' is required to consider the interface as a DAO.
- Line@13: the query annotation provides the select SQL statement that should be executed when the method 'getAllCustomer()' is invoked. You can notice that the select statement at line 13 retrieves all the records in table 'customers' (see entity@line 8) as it does not have the 'WHERE' clause.
- Line@14: the output of 'getAllCustomer' method is LiveData which allows us to observe any changes to the database.
- Line@16: this query is pretty similar to the previous one except it contains a 'WHERE' clause. It retrieves all the customers with a name equal to a value that is provided as an input parameter to the method 'getCustomer'. Please note the colon (:) which is required to specify the name of

the variable that should be used.

- Line@19: the '@Insert' annotation inserts the object that is passed through the method 'addCustomer'
- Line@22: this delete query deletes all the rows that have names equal to the input parameter to the method 'deleteCustomer'
- Line@25: this query deletes all the rows (empty the table) if a call to method 'deleteAllCustomers' occurs

Room Repository

The repository is a Java class that provides an easy and clean API so that the application can access different data sources as depicted below. As you can notice, the repository is managing two different data sources a local SQLite database and a remote data source.




```
1  package com.fit2081.rooms.provider;
2  import android.app.Application;
3  import androidx.lifecycle.LiveData;
4
5  import java.util.List;
6
7  public class CustomerRepository  {
8
9     private CustomerDao mCustomerDao;
10    private LiveData<List<Customer>> mAllCustomers;
11
      CustomerRepository(Application application) {
```

```
12          CustomerDatabase db =
13      CustomerDatabase.getDatabase(application);
14          mCustomerDao = db.customerDao();
15          mAllCustomers = mCustomerDao.getAllCustomer();
16      }
17      LiveData<List<Customer>> getAllCustomers() {
18          return mAllCustomers;
19      }
20      void insert(Customer customer) {
21          CustomerDatabase.databaseWriteExecutor.execute(() ->
22      mCustomerDao.addCustomer(customer));
23      }
24
25      void deleteAll(){
26          CustomerDatabase.databaseWriteExecutor.execute(()->{
27              mCustomerDao.deleteAllCustomers();
28          });
29      }
    }
```

Notes:

- Line@9: declares a reference to the Dao interface which will be used to execute the database operations we have defined earlier.
- Line@10: defines an array list that is used to hold a copy of your data.
- Line@12: this contractor creates a reference to the database that will be used to access the Dao interface. The Dao interface will be used later (line@15) to get the list of customers.
- Line@20: this method inserts a new row (object) into the database. It uses the 'databaseWriteExecutor' to access the database and execute the insert SQL statement.
- Line@24: the same as above. This method uses the 'databaseWriteExecutor' to execute the delete statement.

ViewModel

If a fragment or an activity needs special data or a different way to retrieve the data, then the ViewModel is the best place to implement your logic. In the following AndroidViewModel, all the methods (that will be invoked later by the activity or fragment) call their counterpart methods that are implemented in the Repository.

```
1   package com.fit2081.rooms.provider;
2
3   import android.app.Application;
4
5   import androidx.annotation.NonNull;
6   import androidx.lifecycle.AndroidViewModel;
7   import androidx.lifecycle.LiveData;
8
9   import java.util.List;
10
11  public class CustomerViewModel extends AndroidViewModel {
12      private CustomerRepository mRepository;
13      private LiveData<List<Customer>> mAllCustomers;
14
15      public CustomerViewModel(@NonNull Application application) {
16          super(application);
17          mRepository = new CustomerRepository(application);
18          mAllCustomers = mRepository.getAllCustomers();
19      }
```

```
20
21      public LiveData<List<Customer>> getAllCustomers() {
22          return mAllCustomers;
23      }
24
25      public void insert(Customer customer) {
26          mRepository.insert(customer);
27      }
28      public void deleteAll(){
29          mRepository.deleteAll();
30      }
31
32  }
```

MaiActivity

Now, it's time to execute some database operations. If the activity's controller (i.e. Activity Java file) needs to access the database, it has to create an instance of the view model as shown in the architecture depicted above.

```
1  private CustomerViewModel mCustomerViewModel;
2
3  mCustomerViewModel = new
4  ViewModelProvider(this).get(CustomerViewModel.class);
5        mCustomerViewModel.getAllCustomers().observe(this, newData -> {
6            adapter.setCustomers(newData);
7            adapter.notifyDataSetChanged();
8            tv.setText(newData.size() + "");
         });
```

Notes:

- Line@3: the app uses the ViewModelProvider (An utility class that provides ViewModels for a scope) to get access to the view model.
- Line@4: this line invokes the getAllCustomers to get the list of customers. Because the output of this method is LiveData, the caller has to observe. The observe method invokes the callback method which is provided in the second parameter each time the data gets changed. The new updates can be accessed via the input variable 'newData'

IMPORTANT:

Don't forget to add the following dependencies to your project:

```
1  dependencies {
2      def room_version = "2.5.1"
3
4      implementation "androidx.room:room-runtime:$room_version"
5      annotationProcessor "androidx.room:room-compiler:$room_version"
6
7      // optional - RxJava2 support for Room
8      implementation "androidx.room:room-rxjava2:$room_version"
9
10     // optional - RxJava3 support for Room
11     implementation "androidx.room:room-rxjava3:$room_version"
12
```

```
    // optional - Guava support for Room, including Optional and
ListenableFuture
    implementation "androidx.room:room-guava:$room_version"

    // optional - Test helpers
    testImplementation "androidx.room:room-testing:$room_version"

    // optional - Paging 3 Integration
    implementation "androidx.room:room-paging:2.5.0-alpha01"
}
```

References:

- https://developer.android.com/jetpack/docs/guide
- https://developer.android.com/training/data-storage/room
- https://developer.android.com/jetpack/androidx/releases/room

Last modified: Thursday, 6 April 2023, 3:56 PM