STORAGE

Where to store data?

- Locally
- Cloud
- Cloud + Locally

Local storage options

Name	Data model	Access mode	Best for	Example
SharedPreference*	Key-value	Private	Small amount	State of a game
Preferences (*androidX*)	Key-value	Private	User setting	App settings
Internal file storage	java.io java.nio	Private	• Private data lifetime as app lifetime • Encryption	Large app data
External file storage	Java.io java.nio	Public	Sharing data	Share photo removable SSD
Room DB	Relational	Private	Structured data	Large structured data
Content provider * Name is misleading	Resource + Relational	Public (CRUD) SQL-like	Shared data among apps	Contacts Calendar ..

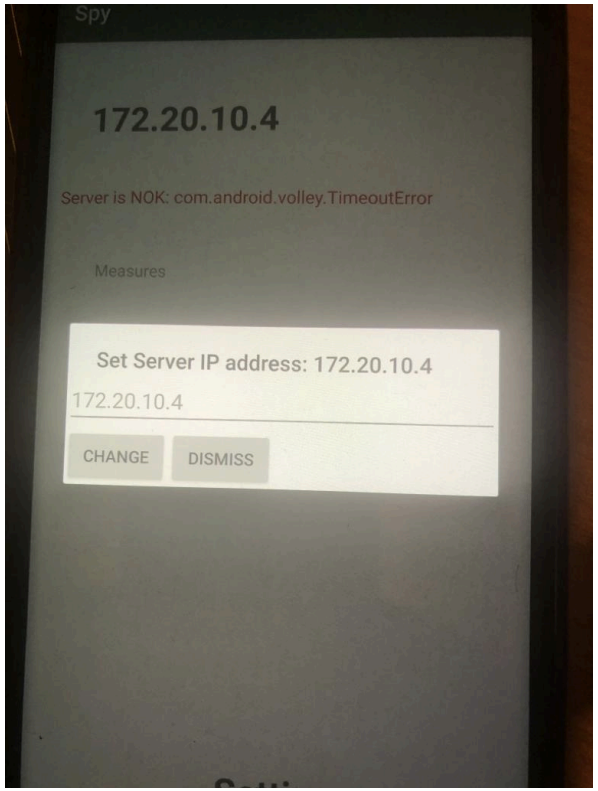
Storage options [key-value]

- **SharedPreference**
  - Allows to store **private primitive** data (booleans, floats, ints, longs, and strings), as Key-value pairs
  - Good option for a small amount of data
    - For example: highest score in a game

- **Preference**
  - Added in *androix*
  - Library to create UI for user's preference

Example

```
setting.setOnClickLis ener { it: View!

    val builder = AlertDialog.Builder( context: this).create()
    builder.setT                    dress: "+ip.toString())
    val dialogView = getLayoutInflater().inflate(R.layout.settings_dialog, root: null)
    dialogView.ip.hint=ip.toString()
    builder.setView(dialogView)

    dialogView.btnOK.setOnClickListener { it: View!
        ip = dialogView.ip.text.toString()

        with (sharedPref.edit()) { this: SharedPreferences.Editor!
            commit() ^with
        }
        Configuration.text=ip
        builder.dismiss()
    }

    dialogView.btnNOK.setOnClickListener { it: View!
        builder.dismiss()
    }

    builder.show()

}
```



Storage options [File]

- **Internal file storage**
  - App private data on the local file system
  - At installation time a **private directory** on the file system is created
- **External file storage**
  - Files saved to the external storage are world-readable
  - Support mount/unmount operations, storage could be **physically removable** (SSD)
    - songs, video
  - …

Example

```
// Although you can define your own key generation parameter specification, it's
// recommended that you use the value specified here.
val keyGenParameterSpec = MasterKeys.AES256_GCM_SPEC
val masterKeyAlias = MasterKeys.getOrCreate(keyGenParameterSpec)

// Creates a file with this name, or replaces an existing file
// that has the same name. Note that the file name cannot contain
// path separators.
val fileToWrite = "my_sensitive_data.txt"
val encryptedFile = EncryptedFile.Builder(
    File(directory, fileToWrite),
    context,
    masterKeyAlias,
    EncryptedFile.FileEncryptionScheme.AES256_GCM_HKDF_4KB
).build()

encryptedFile.openFileOutput().bufferedWriter().use {
    it.write("MY SUPER-SECRET INFORMATION")
}
```

Jetpack security library

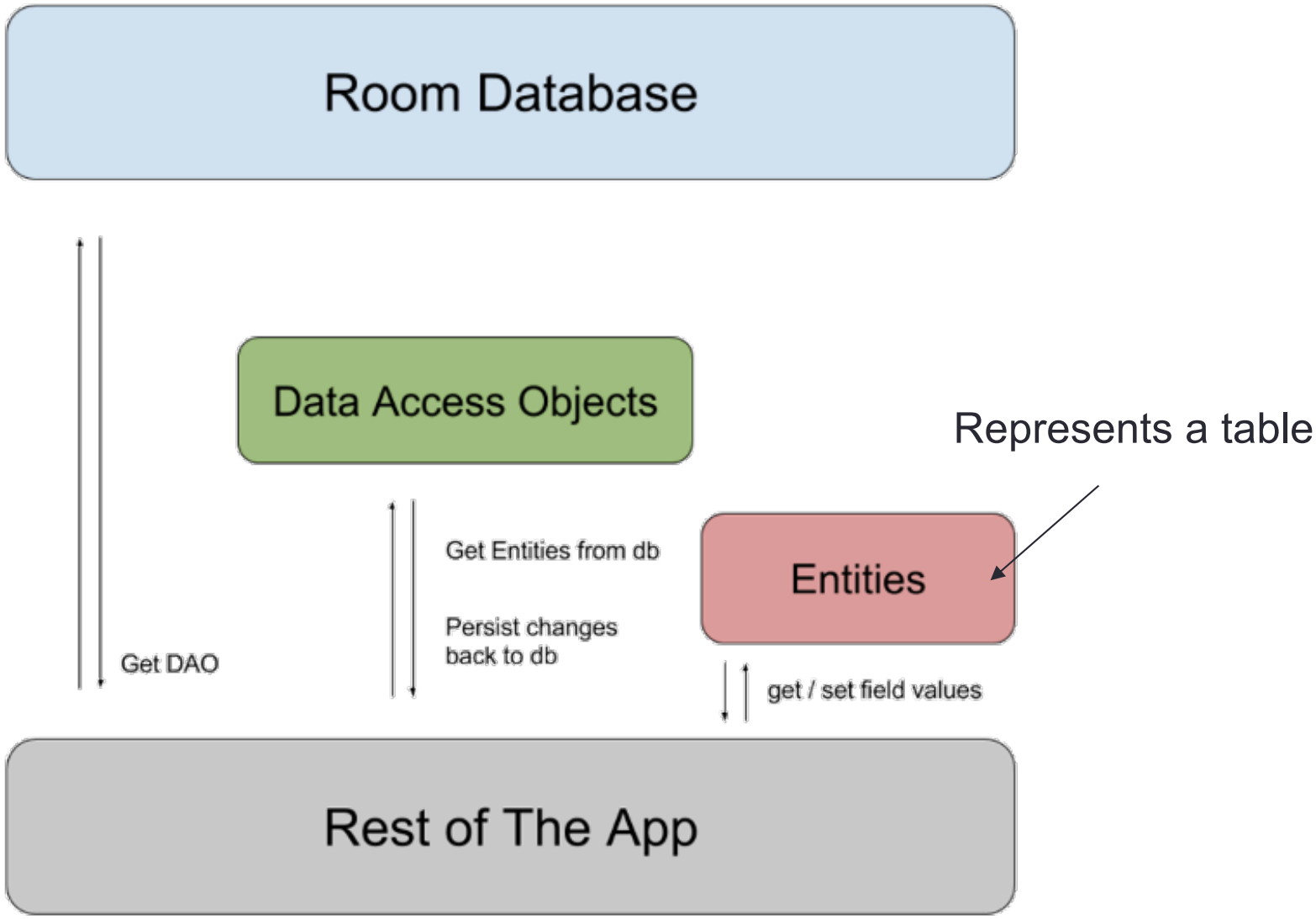
Storage options [database]

- **SQLite**
  - Relational database for private data
  - Recommended access via **Room Persistency Library** (*androix*)

- **Room** provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite

- **Compile time verification** of SQL statement correctness

RoomDatabase and DAO



Example (define a DAO)

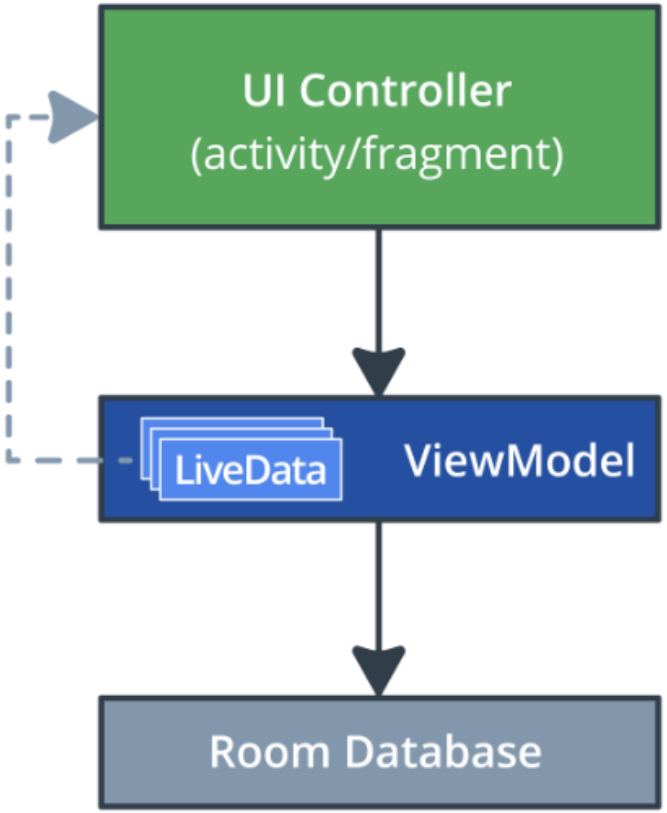
```
@Dao
public interface MovieDAO {

    //CREATE
    @Insert
    void Create(Movie movie); //Create

    //READ ONE MOVIE
    @Query("SELECT * FROM tbl_movies WHERE mid=:mid")
    LiveData<Movie> Read(Integer mid);

    //READ ALL MOVIES
    @Query("SELECT * FROM tbl_movies")
    LiveData<List<Movie>> ReadAll();

    //UPDATE
    @Update
    void Update(Movie movie);

    //DELETE
    @Delete
    void Delete (Movie movie);

    //DELETE ALL
    @Query("DELETE FROM tbl_movies")
    void DeleteAll();

}
```

- Use **annotation @DAO** to mean that a class implements the interface to a DB
- Annotate the methods of the interface with SQL query it will trigger (**@QUERY, ..**)
- The actual query code is generated by the compiler
- SQL statements are then checked at compile time

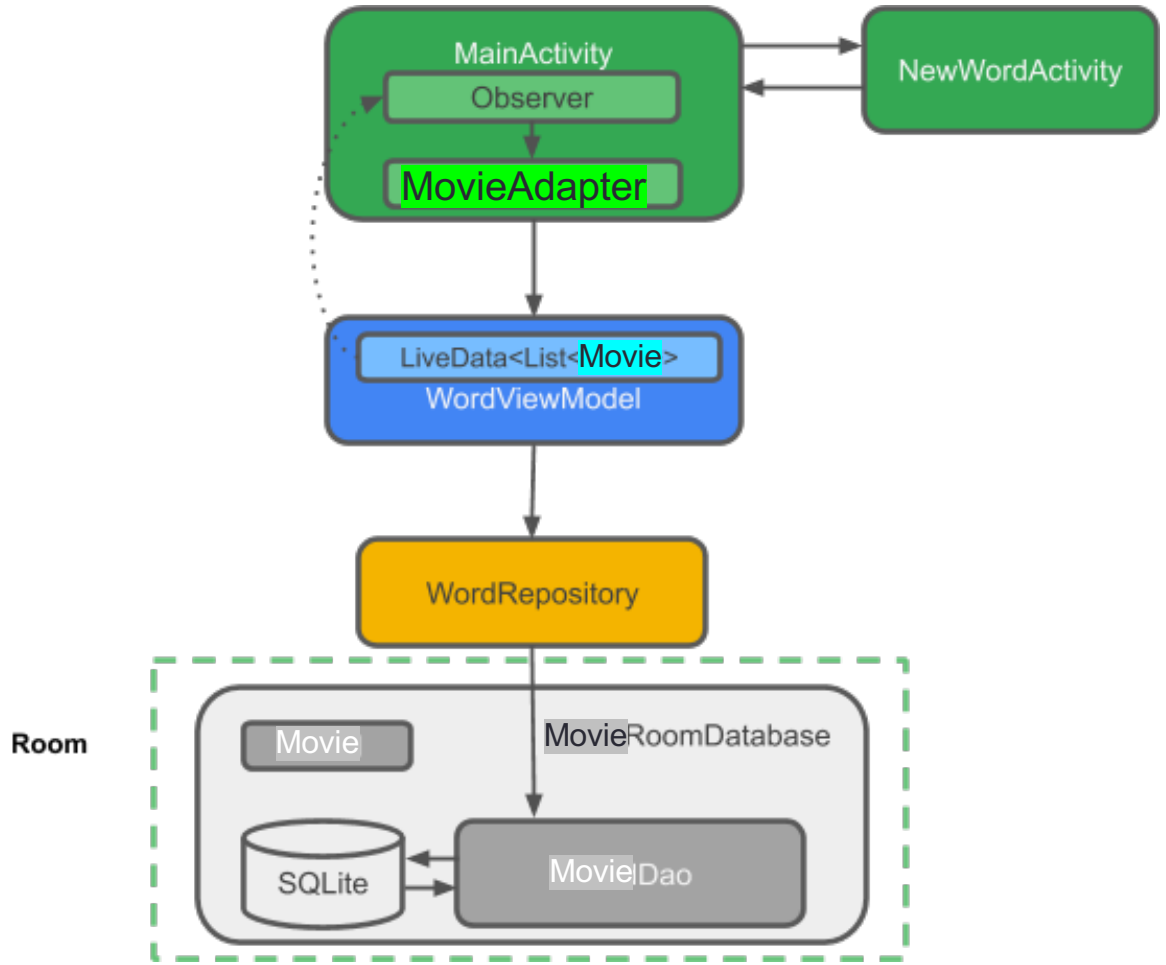
Where Room is used

LiveData notifies observers when data changes

ViewModel makes data lifecycle-aware



Room as Repository



Example

```
@Database(entities = {Movie.class}, version = 1, exportSchema = false)
public abstract class MovieDB extends RoomDatabase

    public abstract MovieDAO movieDAO();
    private static volatile MovieDB INSTANCE;

    static MovieDB getDatabase(final Context context) {
        if (INSTANCE == null) {
            synchronized (MovieDB.class) {
                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                            MovieDB.class,  name: "movie_database3")
                            .build();
                }
            }
        }
        return INSTANCE;
    }

}
```

Entity

```
@Entity
data class Movie(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = title") val title: String?,
    @ColumnInfo(name = ”description") val description: String?
)
```

LiveData

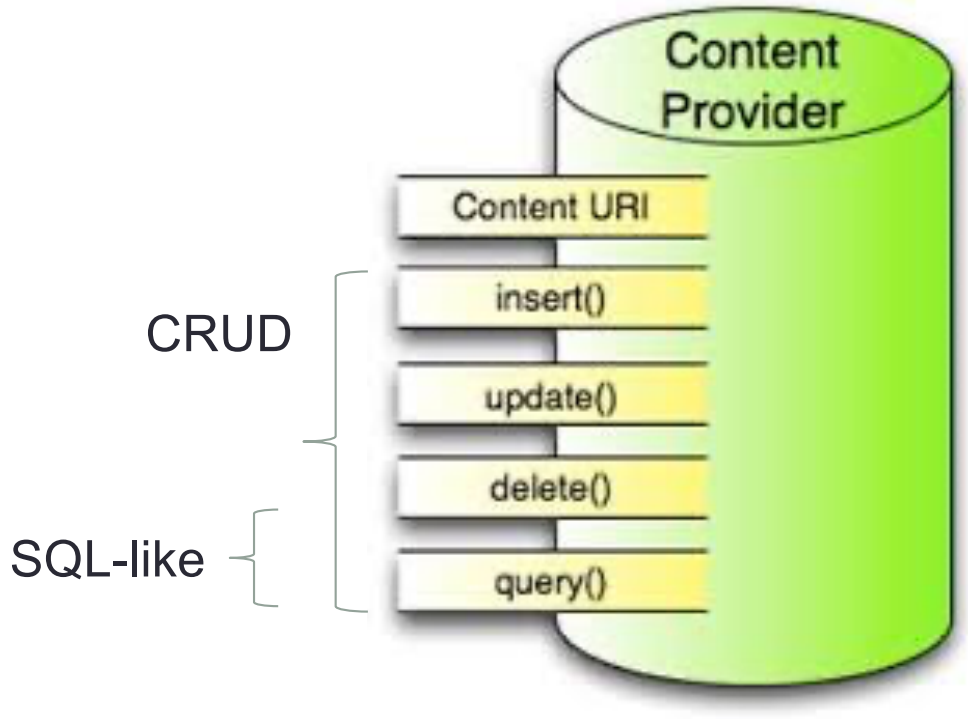
```
class NameActivity : AppCompatActivity() {


private val model: NameViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Other code to setup the activity...

        // Create the observer which updates the UI.
        val nameObserver = Observer<String> { newName ->
            // Update the UI, in this case, a TextView.
            nameTextView.text = newName
        }

        // Observe the LiveData, passing in this activity as the LifecycleOwner and the
observer.
        model.currentName.observe(this, nameObserver)
    }
}
```

LiveData

```
public LiveData<Movie> getMovie(int mid){
    return mdb.movieDAO().Read(mid);
    //return movie;
```

Storage options [Content Provider]

- A repository of information Share information among apps
- Full control of read/write access
- Uniform access interface
  - Can be implemented on a DB or file



Main content providers

- **AlarmClock**
  - Alarms to fire
- **CallLog**
  - contains information about placed and received calls
- **Contact**
  - Stores all information about contacts.
- **Calendar**
  - Data stored in the ‘calendar’ (events, etc.)
- **MediaStore**
  - contains meta data for all available media on both internal and external storage devices.
- **Settings**
  - global system-level device preferences
- **User Disctionary**
  - user defined words for input methods to use for predictive text input
- **Whatsup**

Content Provider URI

- *content://authority/path/id*
  - Content = means one want to access a content provider
  - Authority = name of the content provider
  - Path = 0 or more segments indicating the data to be accessed
  - Id = specific item

- For example:
  - content://com.android.contacts/contacts/directory=0/photo_id/12

Access to a content provider

- A content provider can be accessed lunching an application that manages the provider
- This allows any app to access the content using an **Intent** targeting the provider
- For example
  - **Alarm, Calendar,Contacts**
  - **Whatsup**
  - **…**

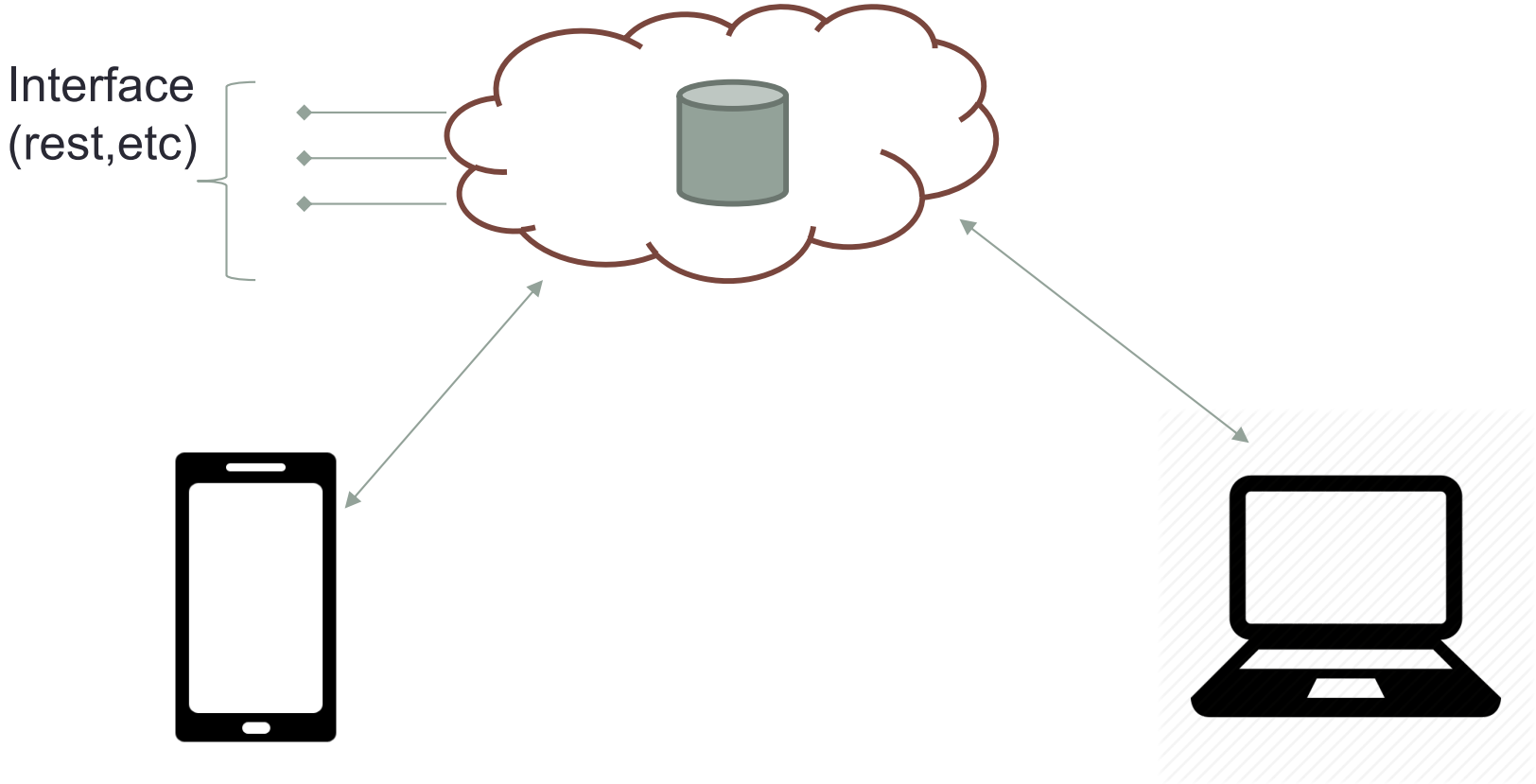
Some example

- *//<uses-permission android:name="com.android.alarm.permission.SET_ALARM" />*
  val i = **Intent**(AlarmClock.***ACTION_SET_ALARM***)
  startActivity(i)

Example

```
Calendar beginTime = Calendar.getInstance();
beginTime.set( year: 2018, month: 1, date: 31, hourOfDay: 9, minute: 30);
Calendar endTime = Calendar.getInstance();
endTime.set( year: 2018, month: 1, date: 31, hourOfDay: 15, minute: 0);
Intent intent = new Intent(Intent.ACTION_INSERT).
                setData(CalendarContract.Events.CONTENT_URI).
                putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME,beginTime.getTimeInMillis()).
                putExtra(CalendarContract.EXTRA_EVENT_END_TIME,endTime.getTimeInMillis()).
                putExtra(CalendarContract.Events.TITLE, value: "MACC Exam").
                putExtra(CalendarContract.Events.EVENT_LOCATION, value: "Room 41,Via Eudossiana");
startActivity(intent);
```

Example: Google calendar Provider



MediaStore provider

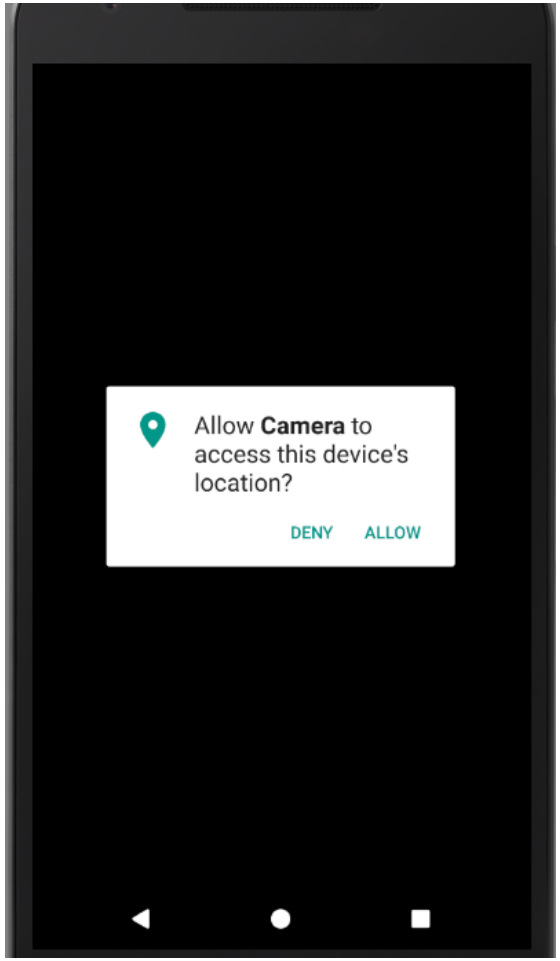
- The Media provider contains meta data for all available media on both internal and external storage devices.

class	MediaStore.Audio Container for all audio content.
class	MediaStore.Files Media provider table containing an index of all files in the media storage, including non-media files.
class	MediaStore.Images Contains meta data for all available images.
interface	MediaStore.MediaColumns Common fields for most MediaProvider tables
class	MediaStore.Video

Example: take a picture

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}

private void takePicture() {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
        startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE);
    }
}
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == RESULT_OK) {
        Bundle extras = data.getExtras();
        Bitmap imageBitmap = (Bitmap) extras.get("data");
        imageView.setImageBitmap(imageBitmap);
    }
}
```

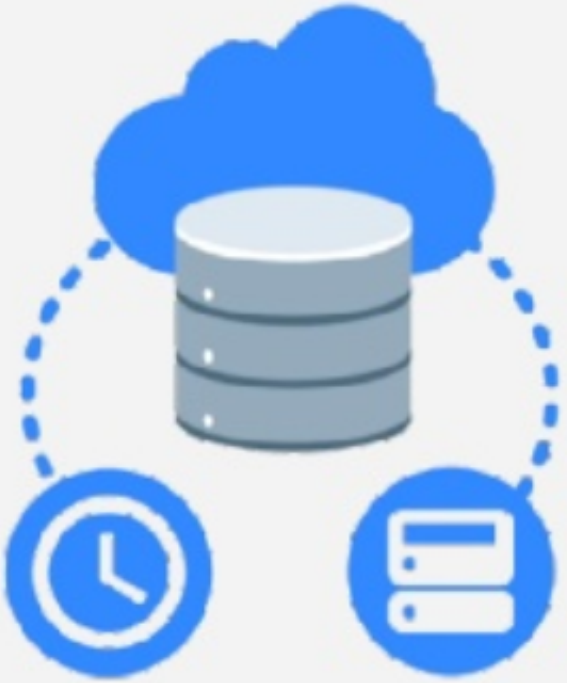
Example: take a picture



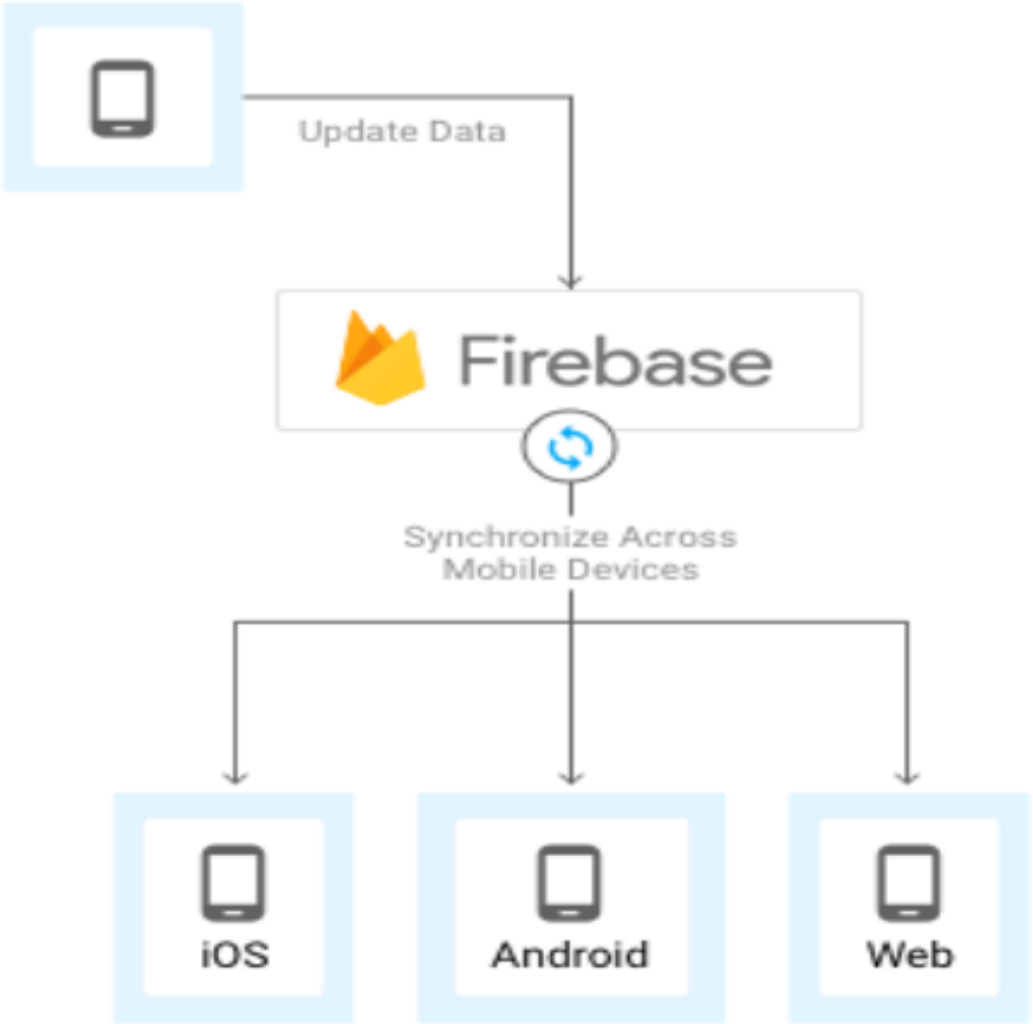
Cloud storage: example

Realtime Database

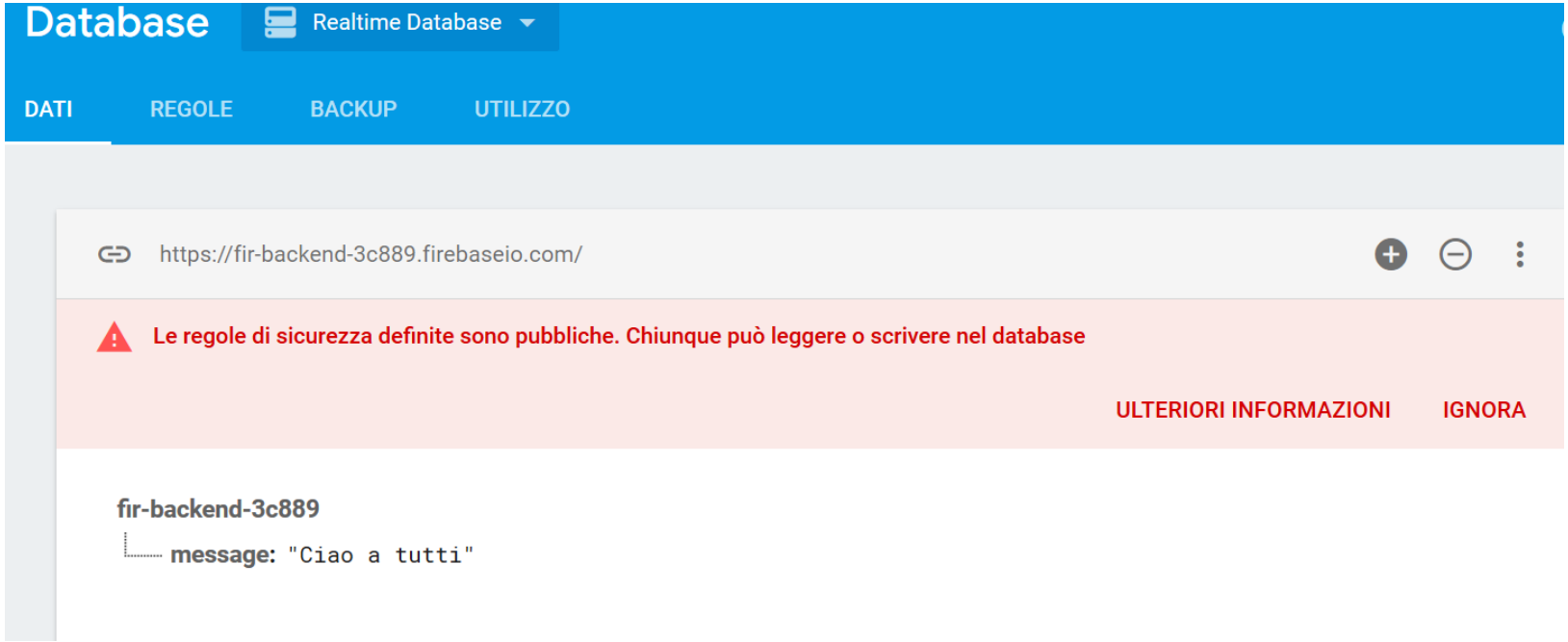
- Cloud-hosted NoSQL database
- Synchronization & conflict resolution
- Access directly from your app



Firebase: Real time db



Exampe


Database
Realtime Database
DATI
REGOLE
BACKUP
UTILIZZO
https://fir-backend-3c889.firebaseio.com/
Le regole di sicurezza definite sono pubbliche. Chiunque può leggere o scrivere nel database
ULTERIORI INFORMAZIONI
IGNORA
fir-backend-3c889
message: "Ciao a tutti"

Example

```
import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ValueEventListener;

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    FirebaseDatabase database = FirebaseDatabase.getInstance();
    myRef = database.getReference("message");
    myRef.setValue("");
    editText = (EditText)findViewById(R.id.messageToSend);

 myRef.addValueEventListener(new ValueEventListener() {
     @Override
     public void onDataChange(DataSnapshot dataSnapshot) {
         String value = dataSnapshot.getValue(String.class);
         ((TextView)findViewById(R.id.messageReceived)).setText(value);

     }

     @Override
     public void onCancelled(DatabaseError error) {
         // Failed to read value
         Log.w(TAG, "Failed to read value.", error.toException());
     }
 });
```