EPFL

Lab on apps development for tablets, smartphones and smartwatches

Week 7: Background tasks and SQLite

Dr. Giovanni Ansaloni, Prof. David Atienza

Ms. Halima Najibi, Ms. Farnaz Forooghifar,
Mr. Renato Zanetti, Mr. Saleh Baghersalimi, Mr. Alireza Amirshahi

Institute of Electrical Engineering (IEL) – Faculty of Engineering (STI)

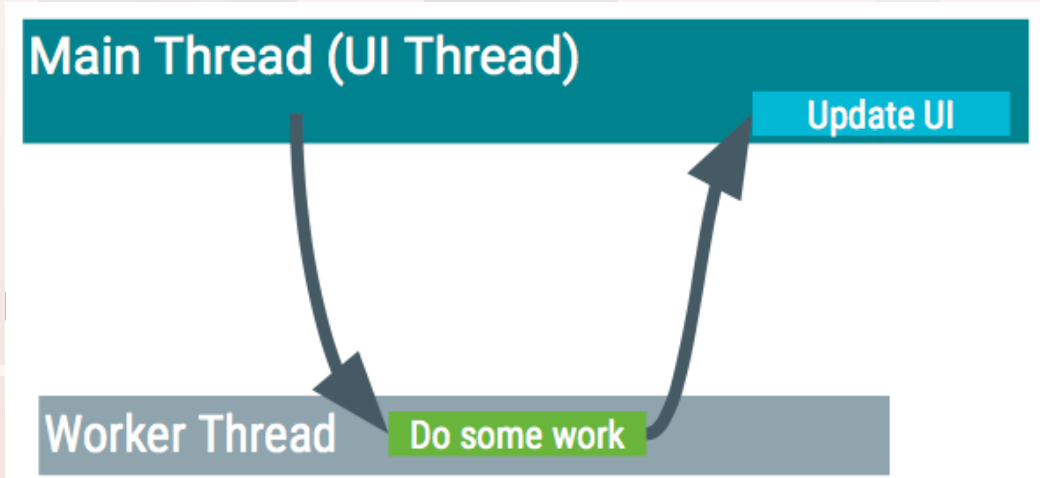
Outline of the class

- **Background tasks**
  - AsyncTask / AsyncTaskLoader
  - Services
    - Started/Bound
    - IntentServices
- SQLite

Quick recap: background tasks

- Reason to use them → The UI must be always fast:
  - Screen is updated every 16ms → UI thread has 16ms to do all the work

- We execute long-running tasks on a background thread
  - Network operations
  - database operations
  - long calculations
  - processing images
  - loading data…



Asynchronous tasks vs Services

- Asynchronous tasks are tightly linked with activities
  - Non-blocking execution of long computation

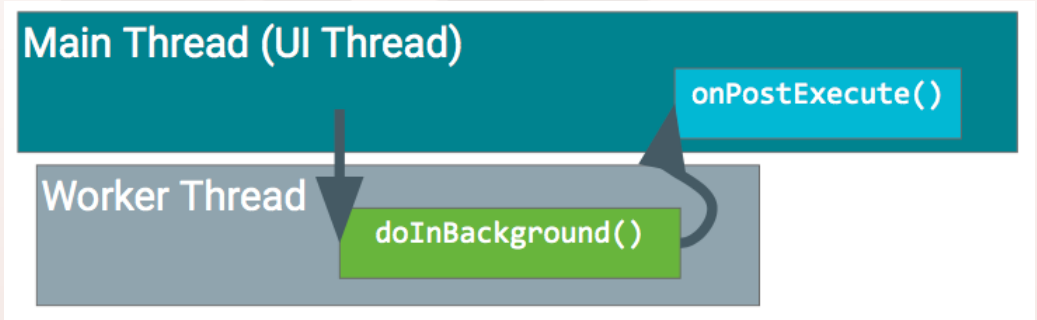
- Services are loosely linked to activities
  - Run even when the activity is destroyed
  - Run independently of an activity → e.g. system services
  - They can initiate events that start an activity

Outline of the class

- Background tasks
  - **AsyncTask / AsyncTaskLoader**
  - Services
    - Started/Bound
    - IntentServices
- SQLite

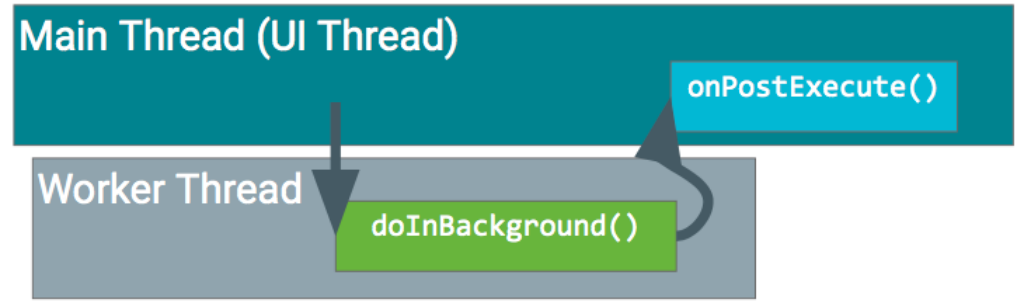
AsyncTask class

- Used to implement basic background task
  - Create a new object of class AsyncTask to take care of a task
- We need to Override two methods:
  - *doInBackground()* → runs on background thread
    - Does all the work that needs to happen on the background
  - *onPostExecute()* → runs on main thread when work is done
    - Processes results and publishes them to the UI
- Two helper methods:
  - *onPreExecute()*
    - Runs on main thread and sets up task
  - *onProgressUpdate()*
    - Runs on main thread
    - Receives calls from background thread



Creating an AsyncTask

1. Extending from the *AsyncTask* class
2. Provide data type sent to *doInBackground()*
3. Provide data type of progress units for *onProgressUpdate()*
4. Provide data type of result for *onPostExecute()*




```
private class MyAsyncTask
1.  extends AsyncTask<String, Integer, Bitmap> {...}
```

2. doInBackground()
3. onProgressUpdate()
4. onPostExecute()

Example: today's lab

```
public class ReadingHeartRateAndLocationAsyncTask extends
        AsyncTask<Void, Void, List<SensorDataEntity>> {

    @Override
    protected List<SensorDataEntity> doInBackground(Void... voids) {
        //...
    @Override
    protected void onPostExecute(List<SensorDataEntity> dbValues) {
        //...
```

Invoking an AsyncTask

- An activity invokes requests an asynchronous task by creating an object of a class extending AsyncTask and running the *execute()* method

- From today's lab

parameters for hrAsyncTask constructor

```
ReadingHeartRateAndLocationAsyncTask hrAsyncTask =
        new ReadingHeartRateAndLocationAsyncTask (…);
hrAsyncTask.execute (…);
```

parameters for doInBackground()

Limitation of AsyncTask

- When device configuration changes, Activity is destroyed
  - AsyncTask cannot connect to Activity anymore
  - New AsyncTask created for every config change → slowdown
  - Old AsyncTasks stay around → memory leaks

AsyncTaskLoader

- Reconnects to Activity after configuration change

- Callbacks implemented in Activity

1. implement the interface *LoaderManager.LoaderCallbacks<Type>*

2. Override the *loadInBackground()* and *onStartLoading()* methods

1.
```
public class MainActivity extends AppCompatActivity
        implements LoaderManager.LoaderCallbacks<String>{
```

2.
```
@Override
public Loader<String> onCreateLoader(int id, final Bundle args) {
    return new AsyncTaskLoader<String>(this) {
        @Override
        public String loadInBackground() {
            //Think of this as AsyncTask doInBackground() method
            return null;
        }

        @Override
        protected void onStartLoading() {
           //Think of this as AsyncTask onPreExecute() method

        }
    };
}
```

AsyncTaskLoader

3. Initialize or restart the loader

4. Override *onLoadFinished()* to do something when the background task finishes
   - similar to onPostExecute()

3.

```
public void methodCalledOnClickButton(View view) {
    // Call getSupportLoaderManager and store it in a LoaderManager variable
    LoaderManager loaderManager = getSupportLoaderManager();
    // Get our Loader by calling getLoader and passing the ID we specified
    Loader<String> loader = loaderManager.getLoader( LOADER_ID );
    // If the Loader was null, initialize it. Else, restart it.
    if(loader==null){
        loaderManager.initLoader( LOADER_ID, myInputsForTask, this );
    }else{
        loaderManager.restartLoader( LOADER_ID, myInputsForTask, this );
    }
}
```

4.

```
@Override
public void onLoadFinished (...) {
    //After getting result we will update our UI here
}
```

Outline of the class

- Background tasks
  - AsyncTask / AsyncTaskLoader
  - **Services**
    - Started/Bound
    - IntentServices
- SQLite

Services

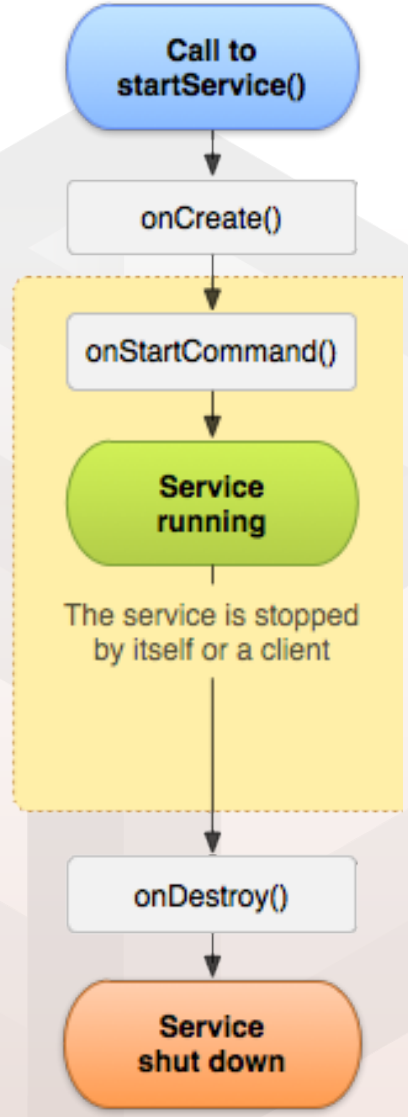
- A service can perform long-running operations in background and does not provide an user interface

- We use them for
  - Network transactions, playing music, perform file I/O, interact with content providers…

- A service extends the **Service** class
  - provides implementation of *onStartCommand()* and *onBind()* callbacks

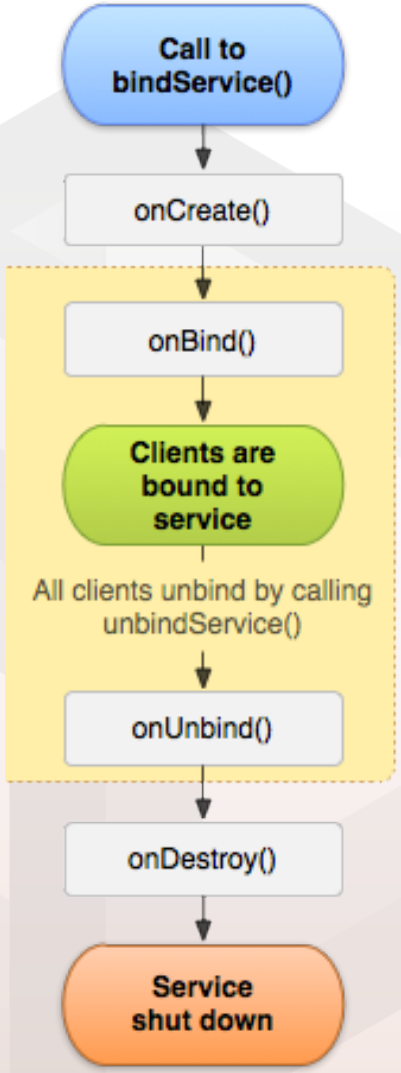
Started Services

- Started
  - Started with *startService(<intent>)*
  - *onCreate()* and *onStartCommand()* are executed
  - Runs indefinitely
    (even if the component who started it is destroyed)
  - … until it stops itself with *selfStop()*
    or is terminated by another via *stopService()*



Bounded Services

- Bound
  - Offers a client-server interface that allows components to interact with the service
  - Clients send requests and get results
  - Started with *bindService()*
  - Ends when all clients unbind



Bounded Services

1. Create an inner class extending from *Binder* containing or referring to public method clients can call

2. Return the binder on *onBind()*

```
public class LocalService extends Service {
    // Binder given to clients
    private final IBinder binder = new LocalBinder();
    // Random number generator
    private final Random mGenerator = new Random();
    /* Class used for the client Binder. */
1.  public class LocalBinder extends Binder {
        LocalService getService() {
            // Return this instance of LocalService to clients
            return LocalService.this;
        }
    }

    @Override
2.  public IBinder onBind(Intent intent) {
        return binder;
    }

    /** method for clients */
    public int getRandomNumber() {
        return mGenerator.nextInt( bound: 100);
    }
}
```

Interacting with bounded Services

1. Acrivities use *bindService()*
   → *onBind()* is executed in the Service

2. Manage service *connection*
   - passed to *bindService()*
   - containing handler to the service *binder*

1.

```
@Override
protected void onStart() {
    super.onStart();
    // Bind to LocalService
    Intent intent = new Intent( packageContext: this, LocalService.class);
    bindService(intent, connection, Context.BIND_AUTO_CREATE);
}
```

2.

```
private ServiceConnection connection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName className,
                                   IBinder service) {
        LocalService.LocalBinder binder = (LocalService.LocalBinder) service;
        mService = binder.getService();
        mBound = true;
    }
    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        mBound = false;
    }
};
```

Interacting with bounded Services

- Call service method

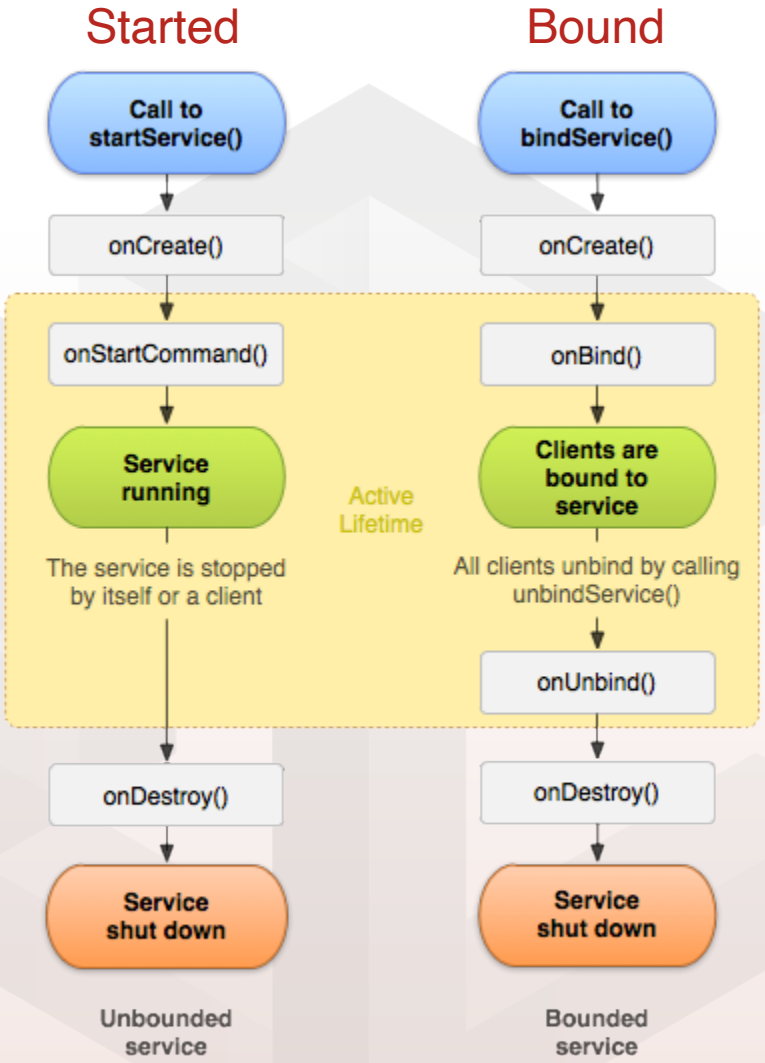
3.

```
public void onButtonClick(View v) {
    if (mBound) {
        int num = mService.getRandomNumber();
        Toast.makeText( context: this,  text: "number: " + num, Toast.LENGTH_SHORT).show();
    }
}
```

Started vs. bounded Services

- Started
  - Started with *startService()*
  - Runs indefinitely (even if the component who started it, is destroyed)
  - … until it stops itself with *selfStop()*
  - or is terminated by another via *stopService()*

- Bound
  - Started with *bindService()*
  - Ends when all clients unbind
  - Offers a client-server interface that allows components to interact with the service
  - Clients send requests and get results



Foreground services

- A type of Service that performs an operation noticeable by the user
- Provides a notification for user interaction
- Use *startForeground()* in Service class

- Useful e.g. for music players


```
public class ForegroundService extends Service {
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Intent notificationIntent = new Intent(this, ExampleActivity.class);
        PendingIntent pendingIntent =
                PendingIntent.getActivity(this, 0, notificationIntent, 0);

        Notification notification =
                new Notification.Builder(this );
            .setContentTitle(getText(R.string.notification_title))
            .setContentText(getText(R.string.notification_message))
            .setSmallIcon(R.drawable.icon)
            .setContentIntent(pendingIntent)
            .setTicker(getText(R.string.ticker_text))
            .build();

        // Notification ID cannot be 0.
        startForeground(ONGOING_NOTIFICATION_ID, notification);
```

IntentServices

- Simple service with simplified lifecycle
- Uses worker threads to fulfill requests
- Stops itself when done
- Ideal for one long task

- We simply create the class and implement the onHandleIntent()

```
public class HelloIntentService extends IntentService {
    public HelloIntentService() { super("HelloIntentService");}

    @Override
    protected void onHandleIntent(Intent intent) {
        // Do some work
        // When this method returns, IntentService stops the service,
    }

}
```

Calling an intentService

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

Stopping a service

- A **started service** must manage its own lifecycle
  - If not stopped, will keep running undefinitely
  - The service must stop itself by calling *stopSelf()*
  - Another component can stop it by calling *stopService()*
- **Bound service** is destroyed when all clients unbound
- **IntentService** is destroyed after *onHandleIntent()* returns

Outline of the class

- Background tasks
  - AsyncTask / AsyncTaskLoader
  - Services
    - Started/Bound
    - IntentServices
- **SQLite**

SQLite

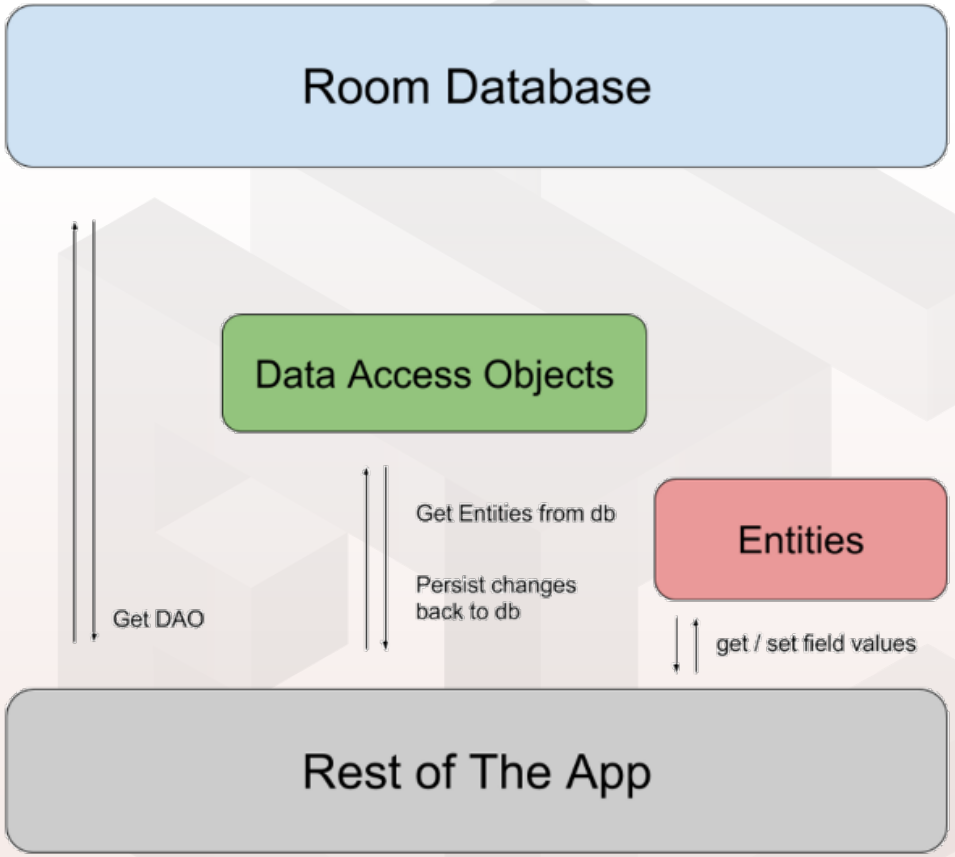
- A database to store structured information
  - Store data in tables of rows and columns (spreadsheet…)
  - Field = intersection of a row and column
  - Rows are identified by unique IDs
  - Column names are unique per table

- Android provides SQL-like database with standard SQL syntax

```
SELECT name FROM table WHERE name = “Luca”
```

Room library (1)

- Room provides an abstraction layer over SQLite
- Three major components
  - Database
    main access point to DB
  - Entity
    represents a table within the database
    define table columns
  - Data Access Objects (DAO)
    methods used for accessing the database



- More details:
  https://developer.android.com/training/data-storage/room

Room library (2)

- Database

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

- Entity

```
@Entity
public class User {
    @PrimaryKey
    private int uid;

    @ColumnInfo(name = "first_name")
    private String firstName;

    @ColumnInfo(name = "last_name")
    private String lastName;

    // Getters and setters are ignored for brevity,
    // but they're required for Room to work.
}
```

- DAO

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND "
           + "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);

    @Insert
    void insertAll(User... users);

    @Delete
    void delete(User user);
}
```

Using the room database

- To use a database, you get an instance of it using the following code:

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database-name").build();
```

- Queries defined in the DAO are executed by calling the corresponding java method

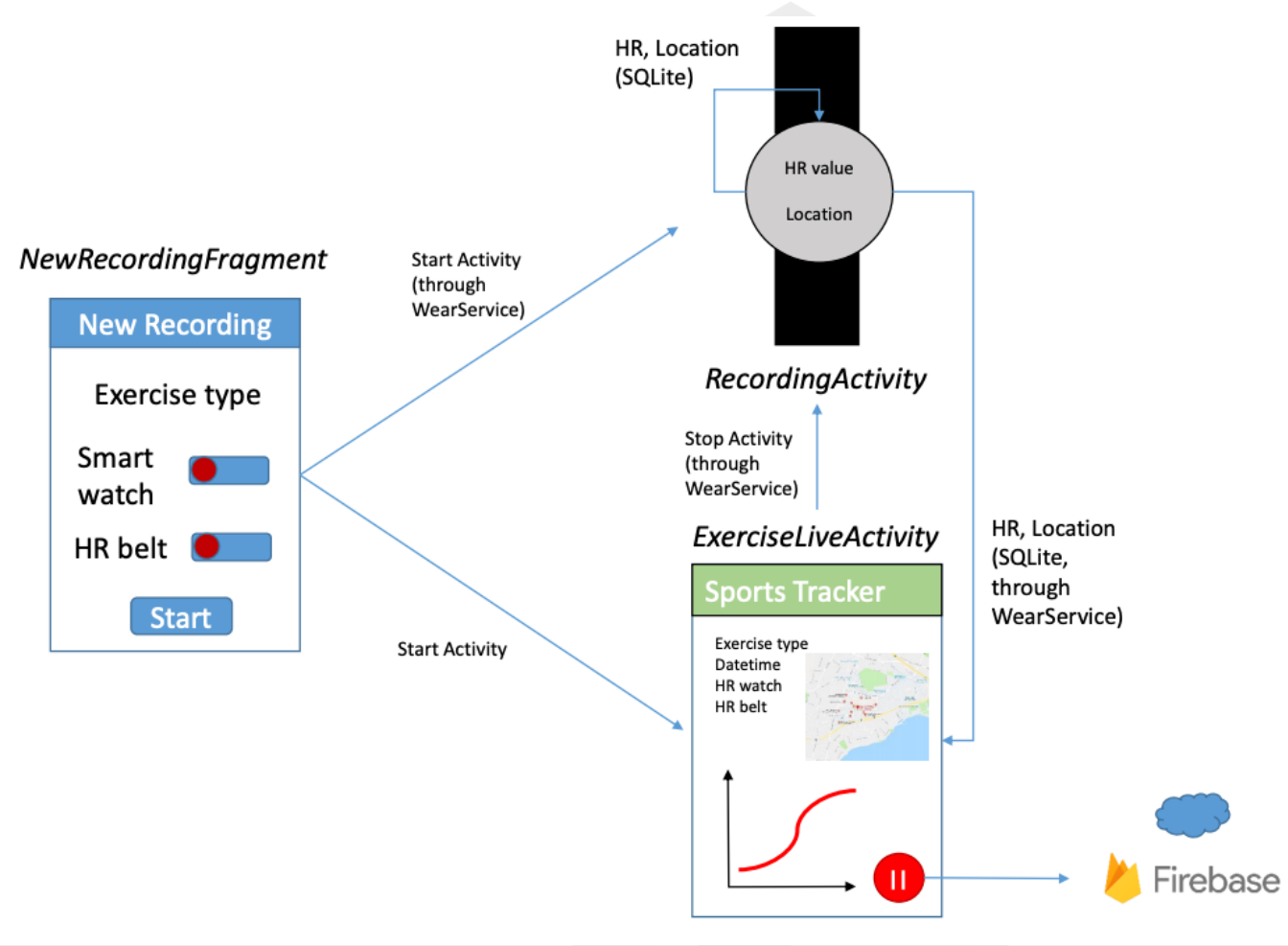
```
user.setName("george");
userDao.insert(user);
```

Commonly used queries

- Insert rows:
  - INSERT INTO table (field1,... fieldN) VALUES (value1,..., valueN);
- Delete rows:
  - DELETE FROM table WHERE column=“value”
- Update rows:
  - UPDATE table SET column=“value” WHERE condition;
- Retrieve rows that meet given criteria
  - SELECT columns FROM table WHERE column="value"

Today's lab

- SQLite
- AsyncTask
- Alarms
- Notifications
- Shared preferences



- Background tasks
  - AsyncTask / AsyncTaskLoader
  - Services
    - Started/Bound
    - IntentServices
- SQLite

Questions?