Mobile Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Android Persistence using Realm

Agenda & Goals

- Be aware of the different approaches to data persistence in Android Development
- Be able to work with the **`SQLiteOpenHelper`** & **`SQLiteDatabase`** classes to implement an SQLite database on an Android device (to manage our Donations)
- Be able to work with **`Realm`** to implement a noSQL database on an Android device (again, to manage our Donations)
- Be able to work with **`SharedPreferences`** to manage our Login & Register screens

Data Storage Solutions *

- **Shared Preferences**
  - Store private primitive data in key-value pairs.
- Internal Storage
  - Store private data on the device memory.
- External Storage
  - Store public data on the shared external storage.
- **SQLite Databases**
  - Store structured data in a private database.
- Network Connection
  - Store data on the web with your own network server.

Data Storage Solutions *

❑ Bundle Class

- A mapping from String values to various `Parcelable` types and functionally equivalent to a standard `Map`.
- Does not handle Back button scenario. App restarts from scratch with no saved data in that case.

❑ File

- Use `java.io.*` to read/write data on the device's internal storage.

❑ Realm Databases

- Store non-structured data in a private database.

realm

What is it?

What is Realm?

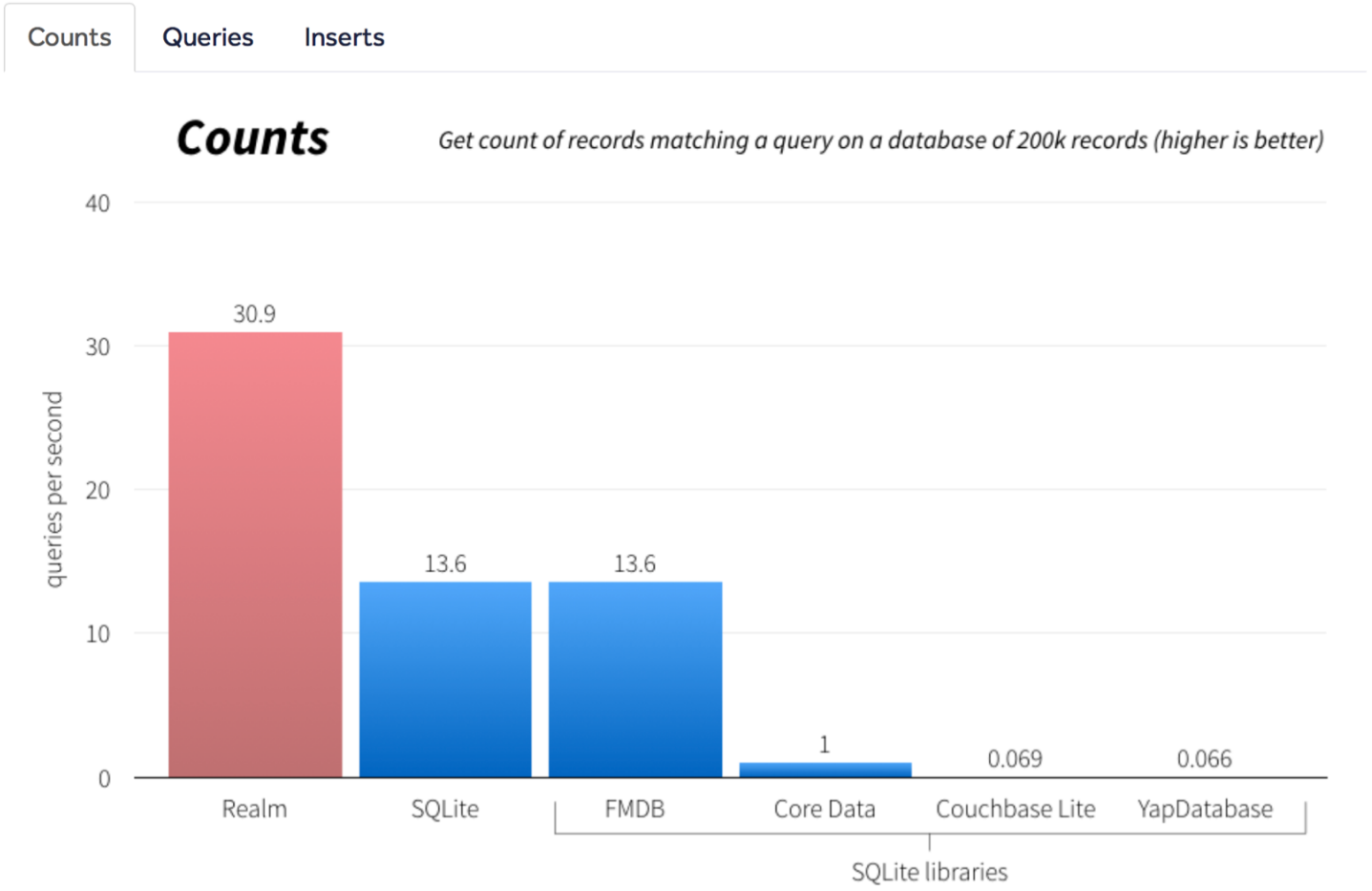
- Modern database written for mobile constraints
- Founded by Alexander Stigsen and Bjarne Christiansen (former Nokia emp.) and launched in 2014
- Available for Android and iOS
  - replacement for SQLite and/or Core Data
- Fast & Scalable
- Object-based
  - queries return objects & relationships to other objects
- Open Source
- Second most-deployed mobile database in the world (03/2018)

Why use it?

- Faster than SQLite (way faster…)
- Easy to use
- Object conversion
- It’s free
- Very good documentation and support
- Thread Safe
- Supports encryption

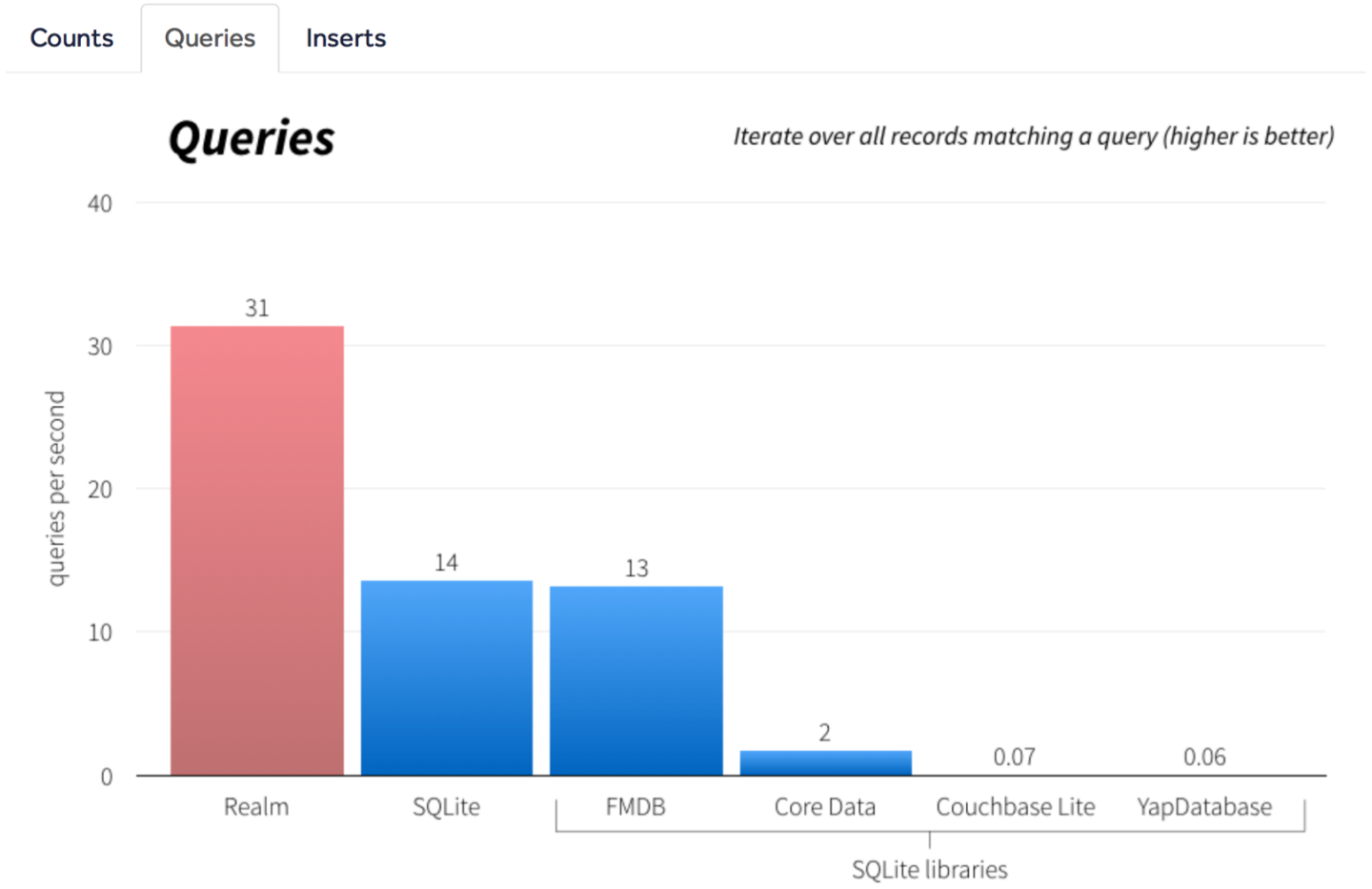
Why use it?

- Faster than SQLite (Counts)



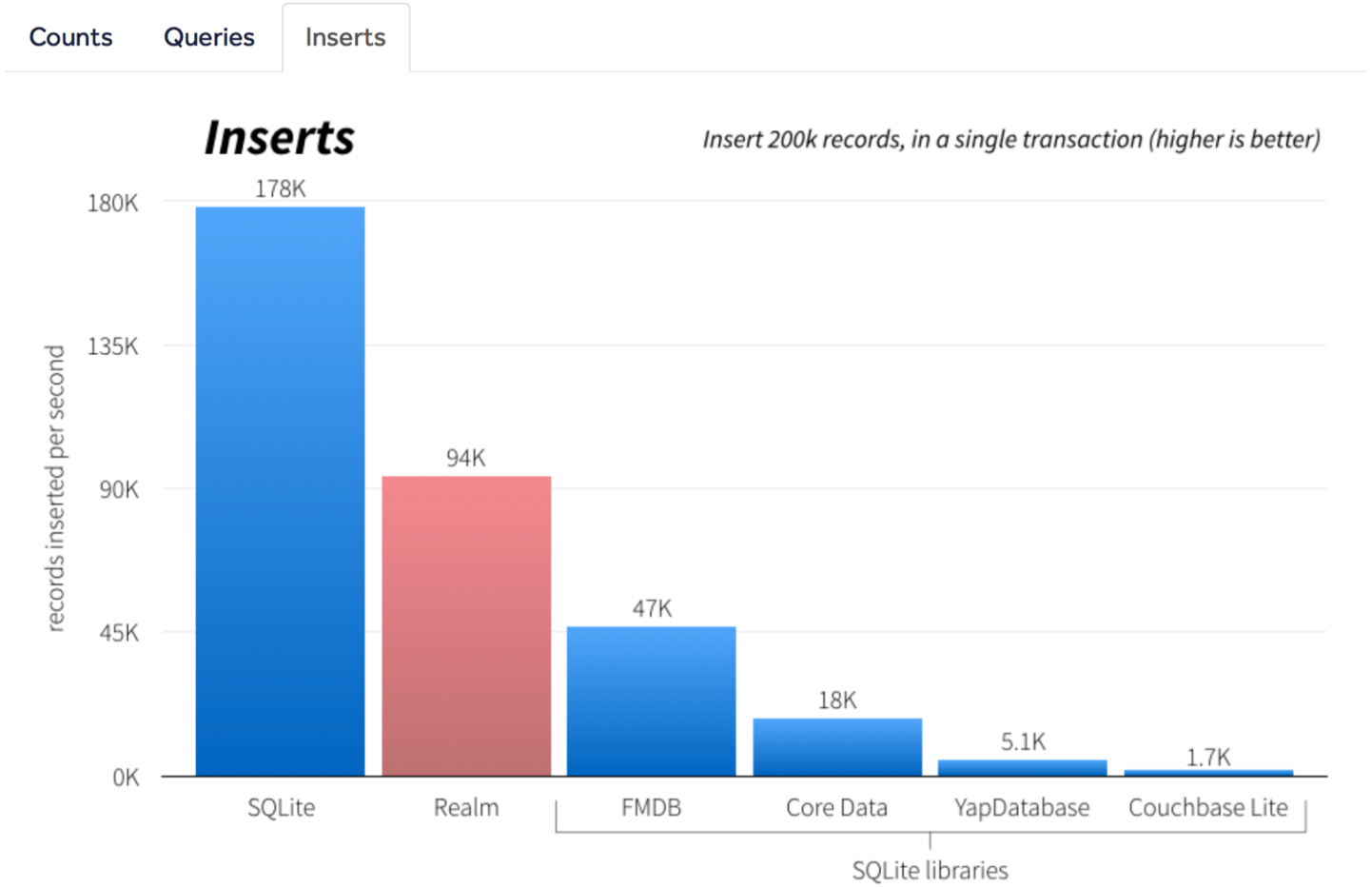
Why use it?

- Faster than SQLite (Queries)



Why use it?

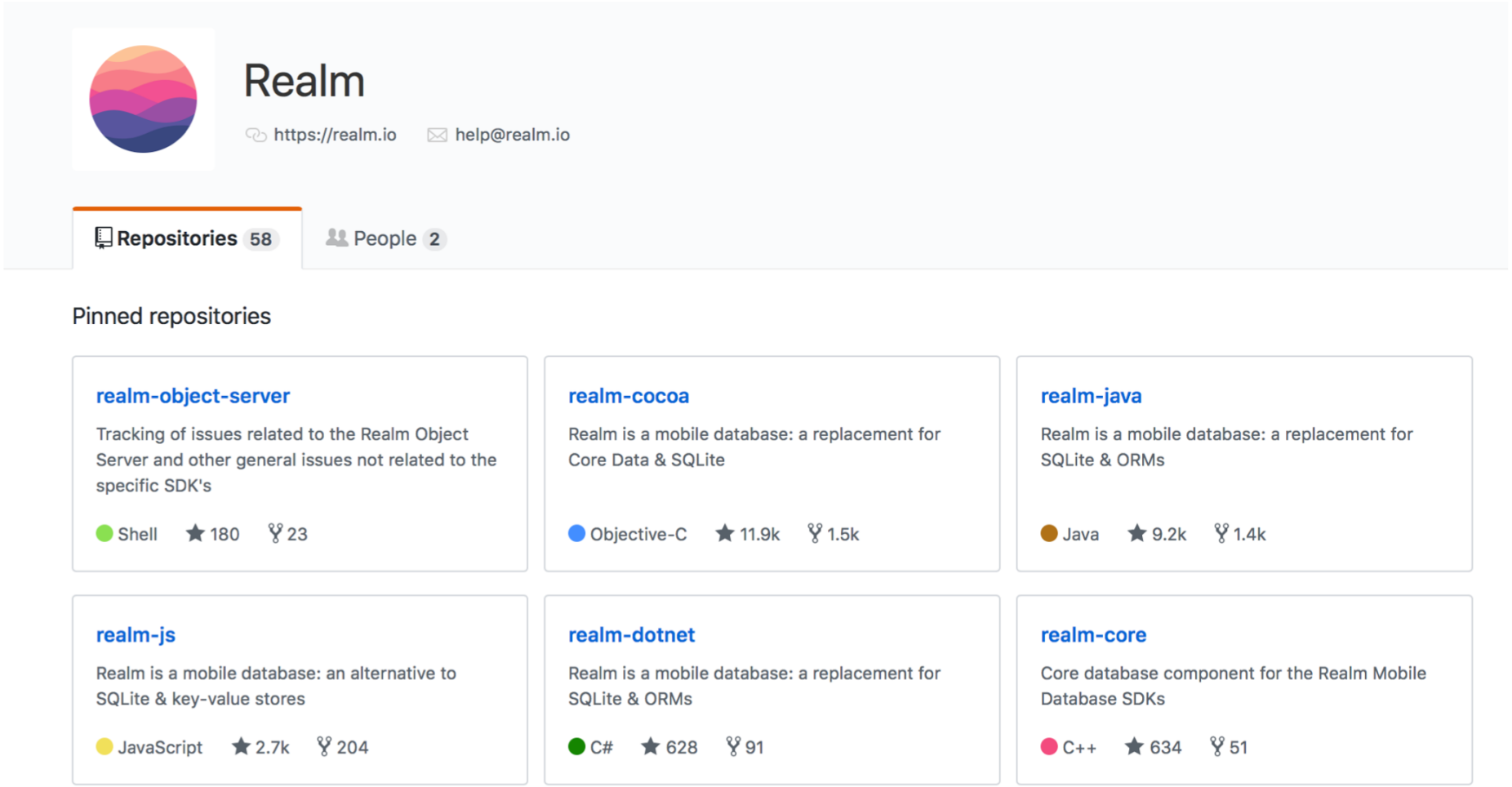
- Faster than SQLite (Inserts)



Source: realm.io, though their benchmark code is available for download and scrutiny.

Why use it?

- Available in Multiple Languages (03/2018)



Why use it?


amazon
Google
hipmunk
STARBUCKS
ebay
REI co-op
NETFLIX
intel
HYATT
BBC
GoPro
SAP
L'ORÉAL
adidas
Alibaba
AVIS
Dropbox
cisco
Walmart
Virgin
NIKKEI
IBM
LINE
intuit
Budweiser
zynga
SoftBank
SIEMENS
SONY
zipcar
AstraZeneca
imgur

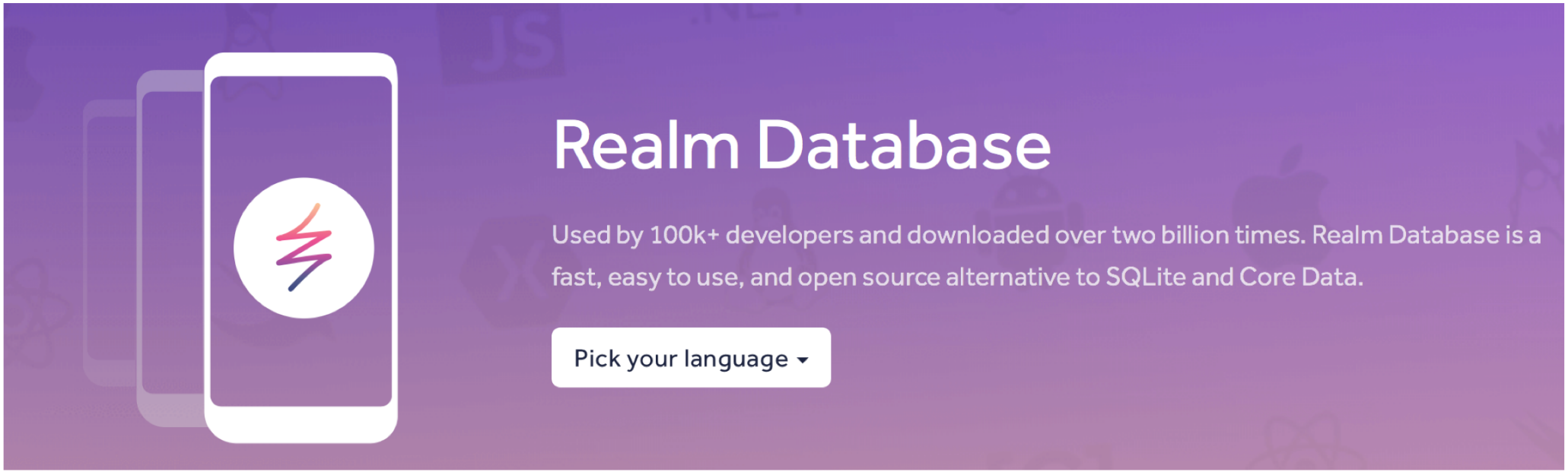
Why use it?

- Suitable for common Mobile Database Use Cases like…
  - Local Storage
    - Ex: grocery list, health tracker, donations list
  - Cache for remote data
    - Ex: search history, donations on a server
  - Pre-loaded Data
    - Ex: Recipe book, trivia game, dummy donations

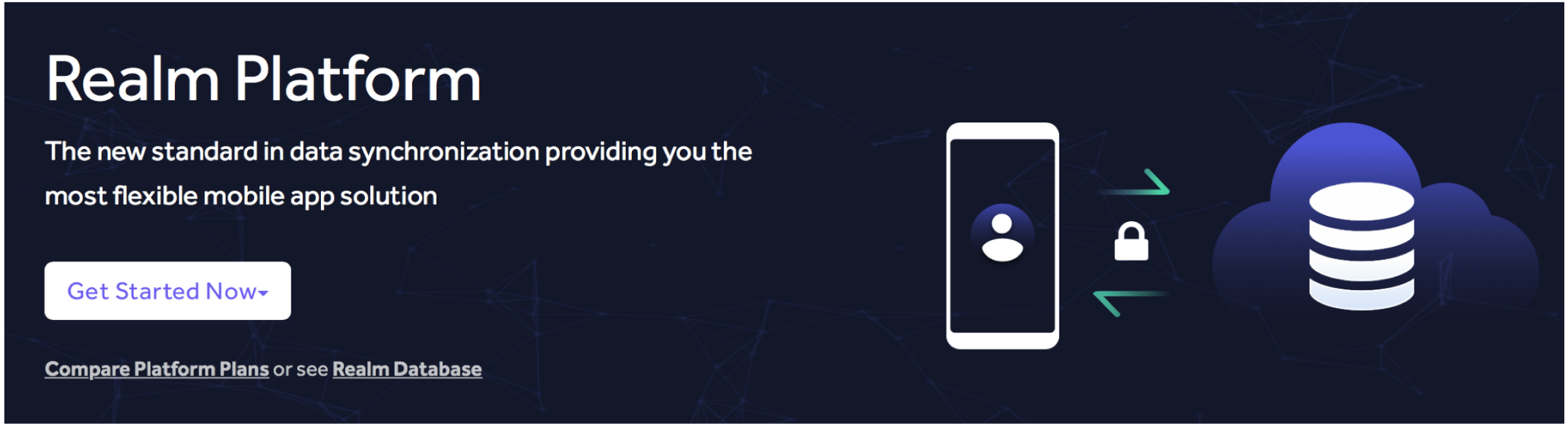
Why use it?

- Suitable for common Mobile Database Use Cases
  - Local Storage
    - define your objects, save them in 3 lines of code, from anywhere (thread independent)
  - Cache for remote data
    - instantiate objects from NSDictionary (iOS), easy insert-or-update methods (using primary keys)
  - Pre-loaded Data
    - Realm files are small, migrating schema (making changes to your db objects) is easy

Why use it? (their Promo stuff ☺)



Why use it?


Realm Platform
The new standard in data synchronization providing you the most flexible mobile app solution
Get Started Now
Compare Platform Plans or see Realm Database

Why use it?

Offline-First is the new standard

With Realm Platform's “live object” approach and embedded database, your apps can deliver a great experience — with or without a signal. Realm's realtime sync functionality ensures that data is accessible and conflicts are resolved when signal connectivity is restored.

Read the white paper

Why use it?

Mobilize your legacy databases (SQL, Postgress, Core Data, etc)

Your app's data is stuck inside existing databases that pre-date the realtime demands of modern mobile apps. Realm's data middleware enables you to automatically handle two-way data sync between legacy systems and your app.

Watch the Webinar

Why use it?

Don't let your REST APIs bog you down.

Use Realm as a "RESTless" middleware layer. Connect your endpoints to as many services you need, easily scale as your apps grow, and with no client-side networking or serialization code to maintain, your team can focus on more important things.

Read the white paper

Why use it?

Your users wait less with realtime sync.

With Realm's realtime data sync, you can drastically improve your app's performance and keep users happy with reactive apps that always feel "alive. Data as objects means that data changes are synced across all clients and the server — no serialization or networking code required.

Read the white paper

Why use it?

Bring the server closer to your users with edge computing

Edge computing capacity enables you to cache or handle data anywhere — on the server or device, or anywhere in between. The Realm Platform quickly adds data sync, conflict resolution, and event handling into your architecture.

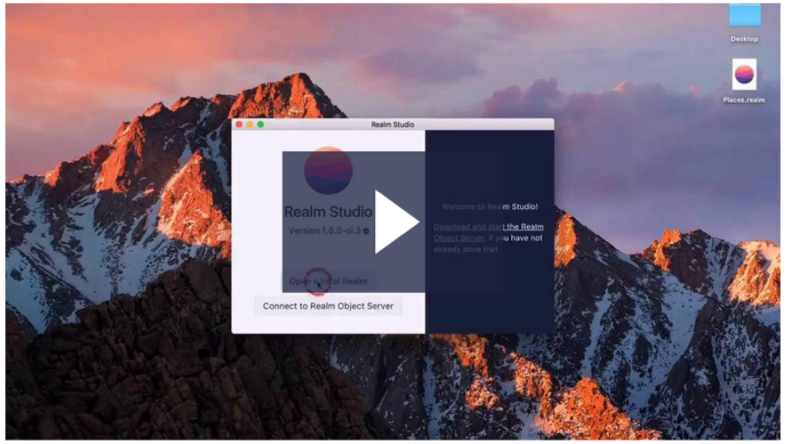
Contact us to learn more

Why use it?

Realm Studio

Realm Studio is our premiere developer tool, built so you can easily manage the Realm Database and Realm Platform. With Realm Studio, you can open and edit local and synced Realms, and administer any Realm Object Server instance. Download it now on macOS, Windows, or Linux.

Download Realm Studio



What is *a* Realm??

- A **Realm** is an instance of a Realm Mobile Database container. Realms can be *local* or *synchronized.*
- A synchronized Realm uses the Realm Object Server to transparently synchronize its contents with other devices. While your application continues working with a synchronized Realm as if it's a local file, the data in that Realm might be updated by any device with write access to that Realm.
- In practice, your application works with any Realm, local or synchronized, the same way, although opening a synchronized Realm requires a User that's been authenticated to the Object Server and that's authorized to open that Realm.

realm

How Does it work?

How it Works – Configuration/Installation

- Step 1: Add the following class path dependency to the **project level** build.gradle file.

```
dependencies {
    classpath 'com.android.tools.build:gradle:3.0.1'
    classpath "io.realm:realm-gradle-plugin:2.2.1"

    // NOTE: Do not place your application dependencies here; they belong
    // in the individual module build.gradle files
}
```

- Step 2: Apply the realm-android plugin to the top of **application level** build.gradle file

```
apply plugin: 'com.android.application'
apply plugin: 'realm-android'
```

How it Works - Models

- Create Realm models by extending the `RealmObject` base class:

```java
// Define your model class by extending RealmObject
public class Dog extends RealmObject {
    private String name;
    private int age;

    // ... Generated getters and setters ...
}

public class Person extends RealmObject {
    @PrimaryKey
    private long id;
    private String name;
    private RealmList<Dog> dogs; // Declare one-to-many relationships

    // ... Generated getters and setters ...
}
```

```java
// Use them like regular java objects
Dog dog = new Dog();
dog.setName("Rex");
dog.setAge(1);
```

- A Realm Model Class supports `public`, `protected` and `private` fields, as well as custom methods.

How it Works – Initializing & Opening Realms

- Before you can use a Realm in your app, you must initialize it. This only has to be done once.

```
// Initialize Realm (just once per application)
Realm.init(context);

// Get a Realm instance for this thread
Realm realm = Realm.getDefaultInstance();
```

- The `getDefaultInstance` method instantiates the Realm with a default `RealmConfiguration`. (next slide)
- We'll put this code in our DBManager Class, but could easily be placed in an Application Object, or even a Base Activity.

How it Works – Configuring a Realm

- To control how Realms are created, use a `RealmConfiguration` object. The minimal configuration usable by Realm is:

```
RealmConfiguration config = new RealmConfiguration.Builder().build();
```

- That configuration—with no options—uses the Realm file 'default.realm' located in `Context.getFilesDir`.

How it Works – Configuring a Realm

- To use another configuration, you would create a new `RealmConfiguration` object:

```
// The RealmConfiguration is created using the builder pattern.
// The Realm file will be located in Context.getFilesDir() with name "myrealm.realm"
RealmConfiguration config = new RealmConfiguration.Builder()
  .name("myrealm.realm")
  .encryptionKey(getKey())
  .schemaVersion(42)
  .modules(new MySchemaModule())
  .migration(new MyMigration())
  .build();
// Use the config
Realm realm = Realm.getInstance(config);
```

How it Works – Configuring a Realm

- You can have multiple `RealmConfiguration` objects, so you can control the version, schema and location of each Realm independently.

```
RealmConfiguration myConfig = new RealmConfiguration.Builder()
  .name("myrealm.realm")
  .schemaVersion(2)
  .modules(new MyCustomSchema())
  .build();

RealmConfiguration otherConfig = new RealmConfiguration.Builder()
  .name("otherrealm.realm")
  .schemaVersion(5)
  .modules(new MyOtherSchema())
  .build();

Realm myRealm = Realm.getInstance(myConfig);
Realm otherRealm = Realm.getInstance(otherConfig);
```

How it Works – Query Your Realm

- Executing queries on your Realm is very straight forward, for example :

```
// Query Realm for all dogs younger than 2 years old
final RealmResults<Dog> puppies = realm.where(Dog.class).lessThan("age", 2).findAll();
puppies.size(); // => 0 because no dogs have been added to the Realm yet
```

- There are numerous ‘clauses’ available for querying your data. For full details, dive into the RealmQuery API reference.

- between(), greaterThan(), lessThan(), greaterThanOrEqualTo() & lessThanOrEqualTo()
- equalTo() & notEqualTo()
- contains(), beginsWith() & endsWith()
- isNull() & isNotNull()
- isEmpty() & isNotEmpty()

How it Works – Query Your Realm

- To find all users named John or Peter, you would write:

```
// Build the query looking at all users:
RealmQuery<User> query = realm.where(User.class);

// Add query conditions:
query.equalTo("name", "John");
query.or().equalTo("name", "Peter");

// Execute the query:
RealmResults<User> result1 = query.findAll();

// Or alternatively do the same all at once (the "Fluent interface"):
RealmResults<User> result2 = realm.where(User.class)
                                  .equalTo("name", "John")
                                  .or()
                                  .equalTo("name", "Peter")
                                  .findAll();
```

How it Works – Adding to Your Realm *

- Inserting data into your Realm is just as easy, for example :

```
// Persist your data in a transaction
realm.beginTransaction();
final Dog managedDog = realm.copyToRealm(dog); // Persist unmanaged objects
Person person = realm.createObject(Person.class); // Create managed objects directly
person.getDogs().add(managedDog);
realm.commitTransaction();
```

- You can either
  - Persist ‘unmanaged’ objects or
  - Create ‘managed objects

How it Works – Auto-updating Objects *

- `RealmObjects` are live, auto-updating views into the underlying data; you never have to refresh objects.
- Changes to objects are instantly reflected in query results.

```
realm.executeTransaction(new Realm.Transaction() {
    @Override
    public void execute(Realm realm) {
        Dog myDog = realm.createObject(Dog.class);
        myDog.setName("Fido");
        myDog.setAge(1);
    }
});
Dog myDog = realm.where(Dog.class).equalTo("age", 1).findFirst();

realm.executeTransaction(new Realm.Transaction() {
    @Override
    public void execute(Realm realm) {
        Dog myPuppy = realm.where(Dog.class).equalTo("age", 1).findFirst();
        myPuppy.setAge(2);
    }
});

myDog.getAge(); // => 2
```

How it Works – Auto-updating Objects

- This not only keeps Realm fast and efficient, it allows your code to be simpler and more reactive.
- If your Activity or Fragment is dependent on a specific `RealmObject` or `RealmResults` instance, you don't need worry about refreshing or re-fetching it before updating the UI.
- You can subscribe to Realm notifications to know when Realm data is updated.

How it Works – Adapters

- Realm offers abstract utility classes to help bind data coming from `OrderedRealmCollections` (both `RealmResults` and `RealmList` implement this interface) to standard UI widgets.
  - Use `RealmBaseAdapter` with `ListView`.
  - Use `RealmRecyclerViewAdapter` with `RecyclerView`.
- To use the adapters, add the following dependency to the application level build.gradle:

```
dependencies {
    compile 'io.realm:android-adapters:2.1.1'
}
```

Other Notable Features

- Sorting
- Chaining Queries
- Filtering
- **ChangeListeners**
- Transaction Blocks
- **Asynchronous Transactions**
- 1-N, N-M & Inverse Relationships
- Read-Only, In-memory Realms

Current Limitations (03/2018)

- Realm aims to strike a balance between flexibility and performance. In order to accomplish this goal, realistic limits are imposed on various aspects of storing information in a Realm. For example:
  - The upper limit of class names is 57 characters. Realm Java prepends class_ to all names, and the browser will show it as part of the name.
  - The length of field names has a upper limit of 63 character.
  - It is not possible to have two model classes with the same name in different packages.
  - Nested transactions are not supported, and an exception is thrown if they are detected.
  - Strings and byte arrays (byte[]) cannot be larger than 16 MB.
  - Realm models have no support for final and volatile fields. This is mainly to avoid discrepancies between how an object would behave as managed by Realm or unmanaged.
  - If a custom constructor is provided, a public no-arg constructor must also be present.
  - Realm model classes are not allowed to extend any other class than RealmObject.

https://realm.io/docs/java/latest/

realm

Compare & Contrast Donation

Database Setup & Connection *

- SQLite

```
public class DBManager {

    private SQLiteDatabase database;
    private DBDesigner dbHelper;

    public DBManager(Context context) {
        dbHelper = new DBDesigner(context);
    }
```

```
public class DBDesigner extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "donations.db";
    private static final int    DATABASE_VERSION = 1;
    private static final String DATABASE_CREATE_TABLE_DONATION = "create table donations "
            + "(id integer primary key autoincrement,"
            + "amount integer not null,"
            + "method text not null);";

    public DBDesigner(Context context) {
        super(context, DATABASE_NAME,  factory: null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase database) {
        database.execSQL(DATABASE_CREATE_TABLE_DONATION);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        Log.w(DBDesigner.class.getName(),
                msg: "Upgrading database from version " + oldVersion + " to "
                        + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS donations");
        onCreate(db);
    }
}
```

- Realm

```
public class DBManager {

    public Realm realmDatabase;

    public DBManager(Context context) {
        Realm.init(context);

        RealmConfiguration config = new RealmConfiguration.Builder()
                .name("donation.realm")
                .schemaVersion(1)
                .build();

        Realm.setDefaultConfiguration(config);
    }
```

Database Setup & Connection

- SQLite

```
public void open() throws SQLException {
    database = dbHelper.getWritableDatabase();
}
```

- Realm

```
public void open() throws SQLException {
    realmDatabase = Realm.getDefaultInstance();
}
```

Create Your Model

- SQLite

```
public class Donation
{
    public int    id;
    public int    amount;
    public String method;

    public Donation (int amount, String method)
    {
        this.amount = amount;
        this.method = method;
    }

    public Donation ()
    {
        this.amount = 0;
        this.method = "";
    }

    public String toString() { return id + ", " + amount + ", " + method; }
}
```

Create Your Model *

- Realm

```
public class Donation extends RealmObject
{
    @PrimaryKey
    public String id;
    public int    amount;
    public String method;

    public Donation() {}

    public Donation (int amount, String method)
    {
        this.id = UUID.randomUUID().toString();
        this.amount = amount;
        this.method = method;
    }

    @Override
    public String toString() {
        return "Donation{" +
                "id = " + id +
                "amount=$" + amount +
                ", method='" + method + '\'' +
                '}';
    }
}
```

No args Constructor

Universal Unique IDentifier

Create Your Schema

- SQLite

```
public class DBDesigner extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "donations.db";
    private static final int    DATABASE_VERSION = 1;
    private static final String DATABASE_CREATE_TABLE_DONATION = "create table donations "
            + "(id integer primary key autoincrement,"
            + "amount integer not null,"
            + "method text not null);";

    public DBDesigner(Context context) {
        super(context, DATABASE_NAME,  factory: null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase database) {
        database.execSQL(DATABASE_CREATE_TABLE_DONATION);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        Log.w(DBDesigner.class.getName(),
                msg: "Upgrading database from version " + oldVersion + " to "
                        + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS donations");
        onCreate(db);
    }
}
```

Create Your Schema

- Realm

Your Model *IS* your Schema

Reading Data *

SQLite

```
public List<Donation> getAll() {
    List<Donation> donations = new ArrayList<Donation>();
    Cursor cursor = database.rawQuery( sql: "SELECT * FROM donations",
            selectionArgs: null);
    cursor.moveToFirst();
    while (!cursor.isAfterLast()) {
        Donation d = toDonation(cursor);
        donations.add(d);
        cursor.moveToNext();
    }
    cursor.close();
    return donations;
}
```

```
private Donation toDonation(Cursor cursor) {
    Donation pojo = new Donation();
    pojo.id = cursor.getInt( i: 0);
    pojo.amount = cursor.getInt( i: 1);
    pojo.method = cursor.getString( i: 2);
    return pojo;
}
```

Realm

```
public OrderedRealmCollection<Donation> getAll() {
    RealmResults<Donation> result = realmDatabase.where(Donation.class)
                                                 .findAll();
    return result;
}
```

Writing Data *

- SQLite

```
public void add(Donation d) {
    ContentValues values = new ContentValues();
    values.put("amount", d.amount);
    values.put("method", d.method);

    database.insert( table: "donations",  nullColumnHack: null, values);
}
```

- Realm

```
public void add(Donation d) {
    realmDatabase.beginTransaction();
    realmDatabase.copyToRealm(d);
    realmDatabase.commitTransaction();
}
```

Deleting Data *

- SQLite

```
public void delete(Donation d) {
    String whereClause = "id=?";
    String ID = new Integer(d.id).toString();
    String whereArgs[] = {ID};
    database.delete( table: "donations", whereClause, whereArgs);
}
```

- Realm

```
public void delete(String id) {
    realmDatabase.beginTransaction();
    realmDatabase.where(Donation.class)
                 .equalTo( fieldName: "id",id)
                 .findAll()
                 .deleteAllFromRealm();
    realmDatabase.commitTransaction();
}
```

Calculations – SUM *

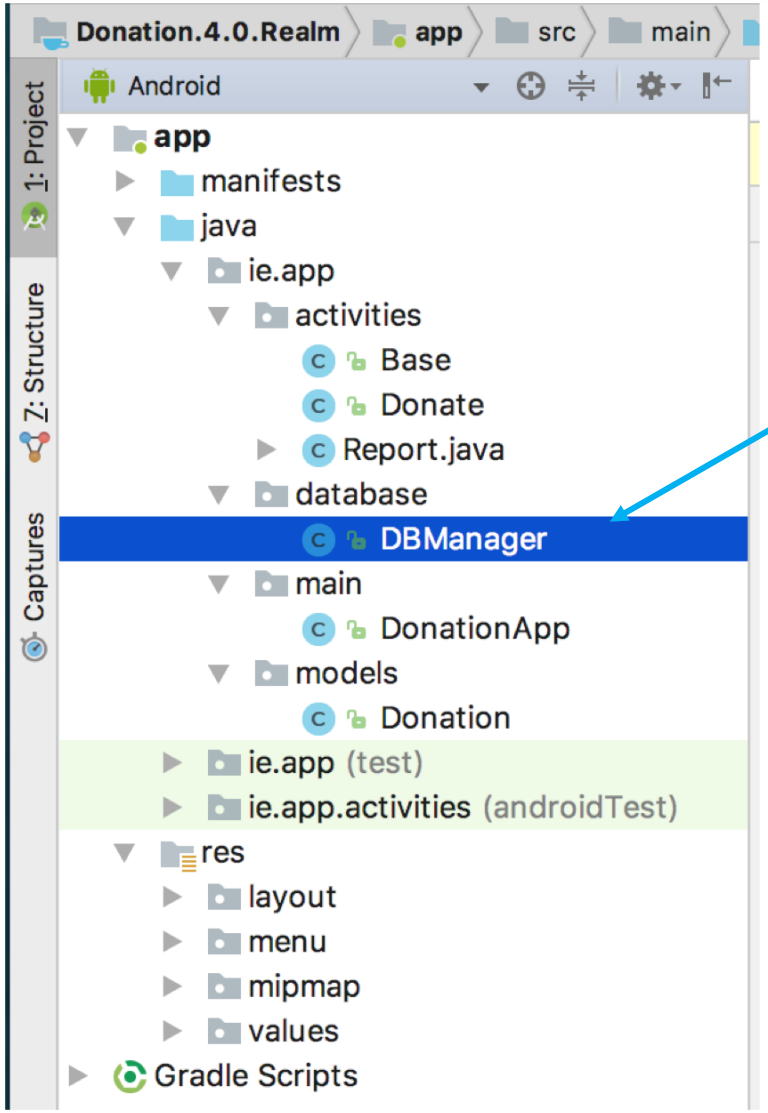
- SQLite

```
public void setTotalDonated(Base base) {
    Cursor c = database.rawQuery( sql: "SELECT SUM(amount) FROM donations",
            selectionArgs: null);
    c.moveToFirst();
    if (!c.isAfterLast())
        base.app.totalDonated = c.getInt( i: 0);
}
```

- Realm

```
public void setTotalDonated(Base base) {
    base.app.totalDonated = realmDatabase.where(Donation.class)
                                         .findAll()
                                         .sum( fieldName: "amount")
                                         .intValue();
}
```

Donation 4.0.Realm – Project Structure (Classes)



- Our Realm Database class
- Xml layouts
- Xml menu
- Xml files for resources
- Xml 'configuration' file

More Reading

- https://realm.io
- https://github.com/realm/presentation/blob/master/slides.md
- http://blog.mallow-tech.com/2017/01/up-and-running-with-realm-for-android/
- https://barta.me/persist-data-android-realm/

Questions?