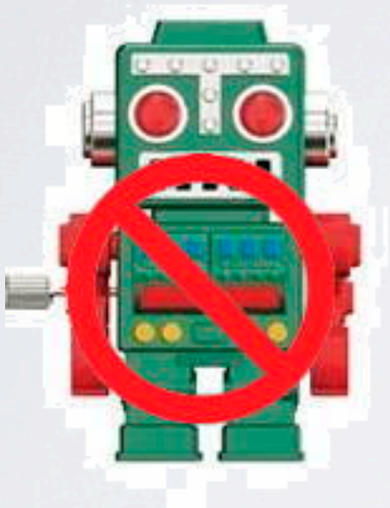
Building Android Applications

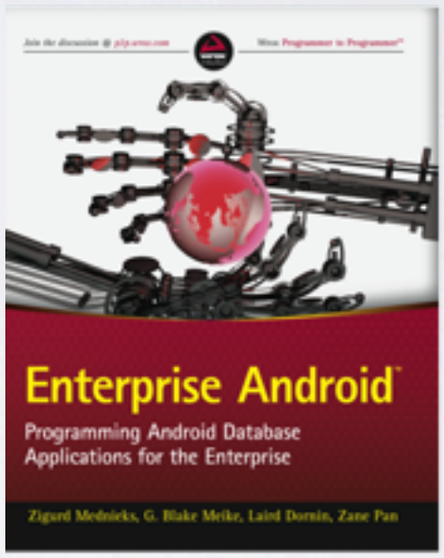
that aren't toys



Blake Meike

Developer, Architect, Android Evangelist

- Speaker at StrangeLoop, OSCON, AnDevCon.
- Author:
  - Enterprise Android (Wiley) .
  - Programming Android (O'Reilly)



blake.meike@gmail.com

twitter: @callmeike

blog: http://portabledroid.wordpress.com/

Android "Desktop Applications"

Open your IDE and start to code:

What do you see?

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Got it!

Activity	=	Application
onCreate	=	public static void main()

Arrgh! Long running tasks!

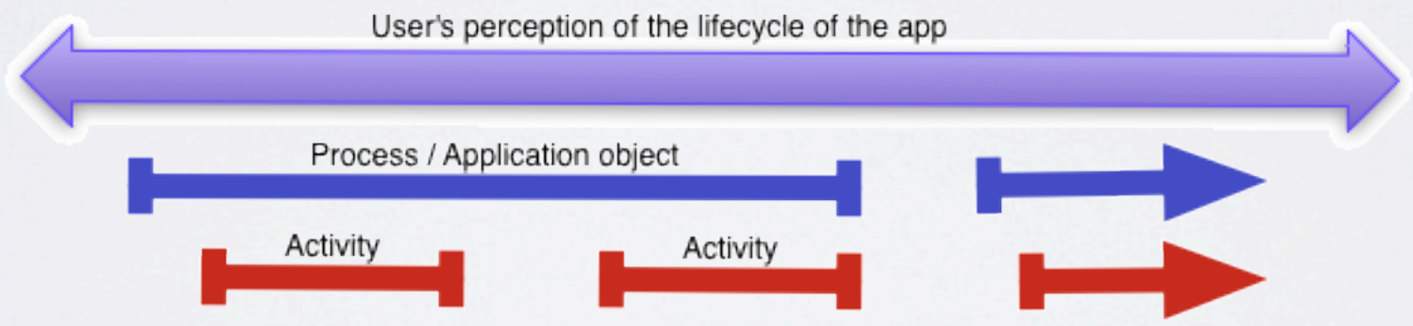
You have experience with	…so you know that
UIs	you can't stall the UI thread!
Java	wildly creating thread drives the garbage collector crazy
Android	AsyncTask!

AsyncTask!

Problem Solved!!!

The Android Lifecycle

- Your application should appear, to the user, to run forever
- You have no control over when your application actually starts and stops!



oom_adj

- A value between -15 and 16
- Larger numbers are more likely to be killed
- <0 is for system services
- A visible Activity is 0
- An invisible Activity is >7
- A running service is < 4

When Android needs space it will kill your app!

kill -9

So... about that AsyncTask?

- The managed object that initiated the request might be gone by the time the request completes: a waste of battery
- The device is powered down (or the battery removed) while the request is in flight. It is lost.
- Android needs space for a new application and kills your app. The request is lost.

The problem is structural

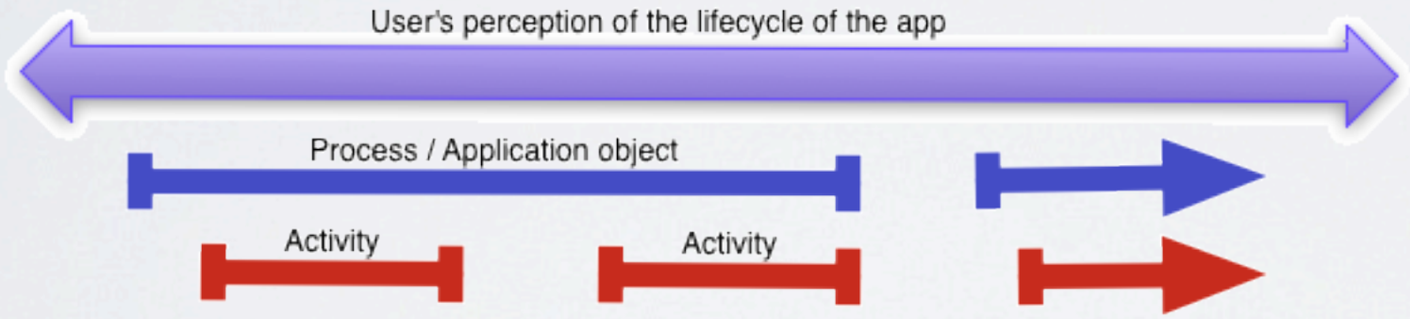
The user perceives, e.g., pushing a "submit" button as a contract.

Representing that contract as an in-memory object is doomed.

Reboot!

Look at the lifecycle again.

Does it remind you of anything?



hint: WebApp.

Android applications are WebApps

Activity	=	Servlet
onCreate	=	init
Service	=	Stateless Session Bean
ContentProvider		DAO

Get your app out of the Activity!

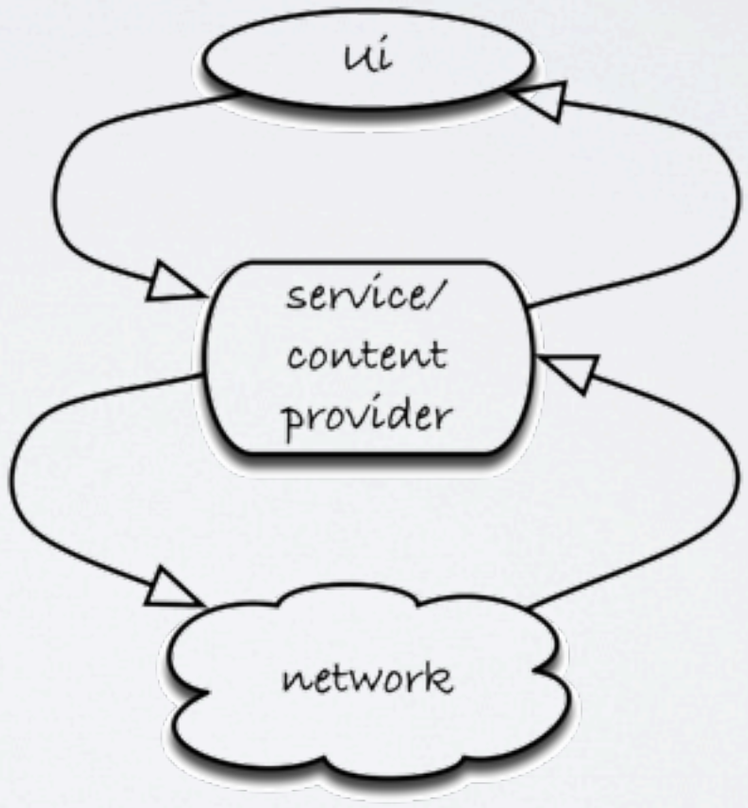
Dobjanschi architecture

http://www.youtube.com/watch?v=xHXn3kg2IQE

No more Async Tasks!

The Figure-8

- The UI is independent of the back-end
- The UI's view is consistent and repeatable
- Keeps promises!



An Architecture for Robust Networking

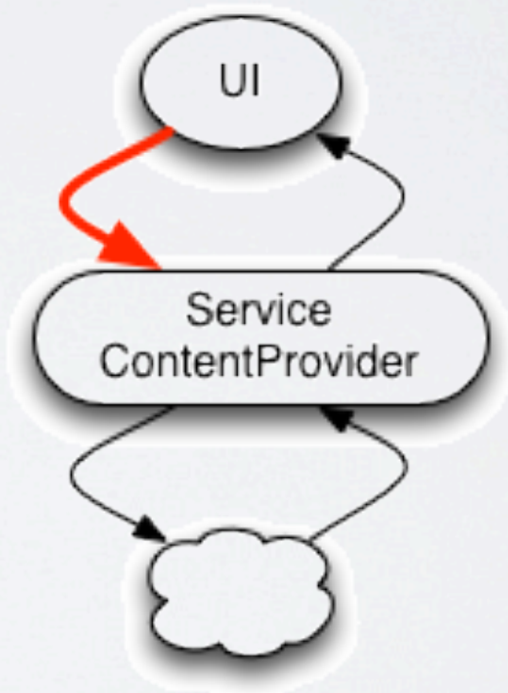
Leverage:

- IntentService
- Loader
- ContentProvider
- SyncAdapter

The Intent Service

An Intent Service is similar to an Async Task:

It is a component with one method that runs on a daemon thread



This is the way a UI should look!

... Back-end concerns belong to someone else

```
void post() {
    String tweet = tweetView.getText().toString();
    if (!checkTweetLen(tweet.length())) { return; }

    tweetView.setText("");

    NetworkService.post(ctxt, tweet);
}
```

Service Helper

The static helper method, in the Service class, marshals the call into an Intent

```
public static void post(Context ctxt, String tweet) {
    Intent i = new Intent(ctxt, NetworkService.class);
    i.setExtra(PARAM_OP, OP_POST);
    i.setExtra(PARAM_TWEET, tweet);
    ctxt.startService(i);
}
```

The first parameter is the “name” of the method.

Other params added as necessary

The UI has been decoupled!!

Handling the Intent

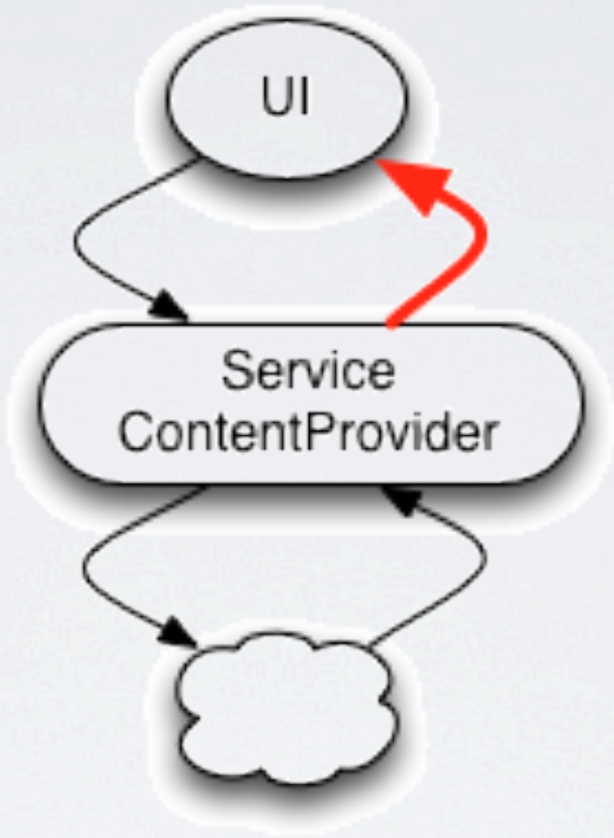
The intent is delivered, asynchronously, to the `onHandleIntent` method, running on a daemon thread.

```
protected void onHandleIntent(Intent intent) {
    int op = intent.getIntExtra(PARAM_OP, 0);
    switch(op) {
        case OP_POST:
            doPost(intent.getStringExtra(PARAM_TWEET, “”));
            break;

        // ...

        default:
            throw new IllegalArgumentException("Unrecognized op: " + op);
    }
}
```

Why is this an Improvement?

- Elegant Architecture: All of the “back end” code is out of the Activity.
- Reliable: An IntentService is running at oom_adj ~ 3 and less likely to be killed.

Looking at Loaders



Using the Loader Manager

Activity registers with the Manager

```
public class MyActivity
    extends Activity
    implements LoaderCallbacks<Cursor>
{
    protected void onCreate(Bundle state) {
        // ...

        getLoaderManager().initLoader(
            TIMELINE_LOADER,
            null,
            this);
    }
}
```

The Loader Manager gets a Cursor

Manager calls back, asynchronously:

... first to get the Loader

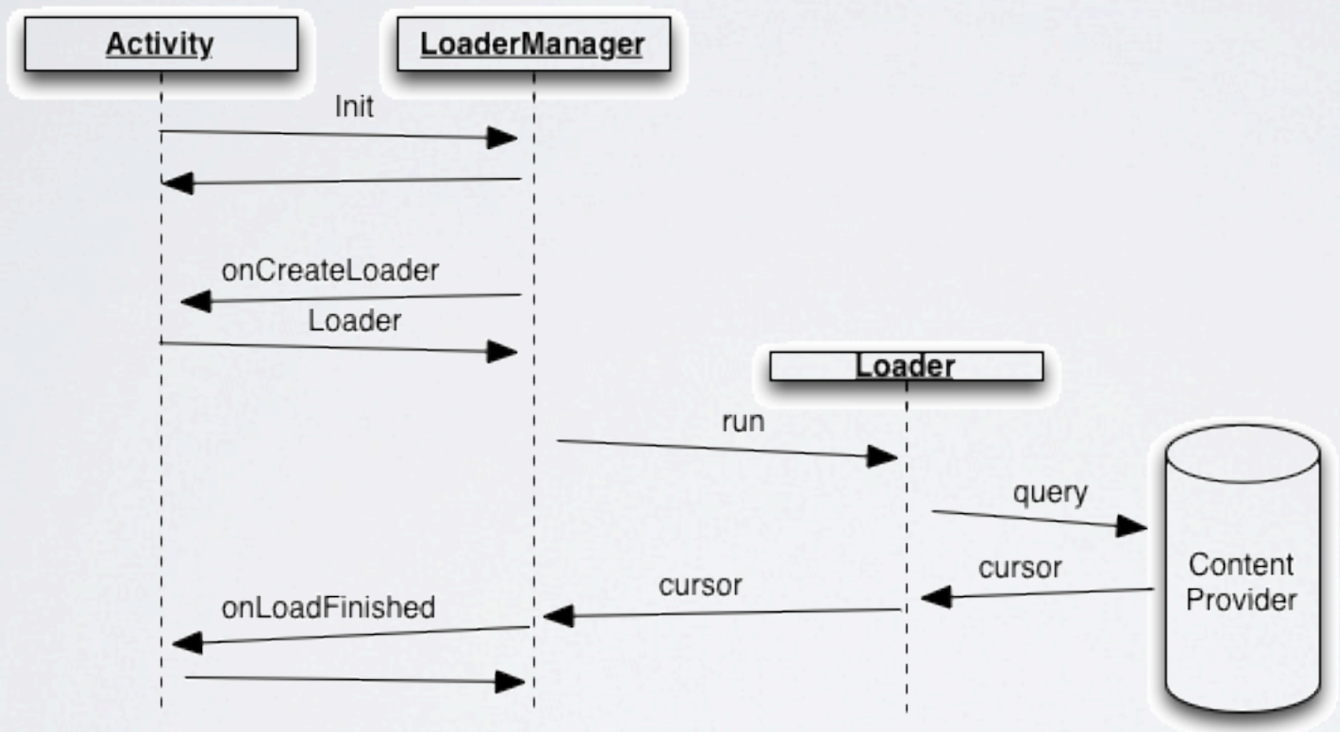
```
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    return new CursorLoader(
        ctxt,
        Contract.URI,
        null, null, null, null);
}
```

... and again to hand off the Cursor

```
public void onLoadFinished(Loader<Cursor> l, Cursor c) {
    ((SimpleCursorAdapter) getListAdapter()).swapCursor(c);
}
```

Why so complicated?

We just want a cursor!
Why all the fuss?



Magic!

The cursor was registered as a URI listener

```
public Cursor query(
    Uri uri,
    String[] proj,
    String sel, String[] selArgs,
    String sort)
{
    // ...

    Cursor c = qb.query(getDb(), proj, sel, selArgs, null, null, sort);

    c.setNotificationUri(getContext().getContentResolver(), uri);

    return c;
}
```

The LoaderManager, in turn, listens to the cursor

Magic!

... and when a change occurs, it gets notified

```
public Uri insert(Uri uri, ContentValues row) {
    // ...

    long id = getDb().insert(
        YambaDbHelper.TABLE_POSTS,
        null,
        COL_MAP_POSTS.translateCols(row));

    if (0 >= id) { return null; }

    uri = uri.buildUpon().appendPath(String.valueOf(id)).build();
    getContext().getContentResolver().notifyChange(uri, null);

    return uri;
}
```

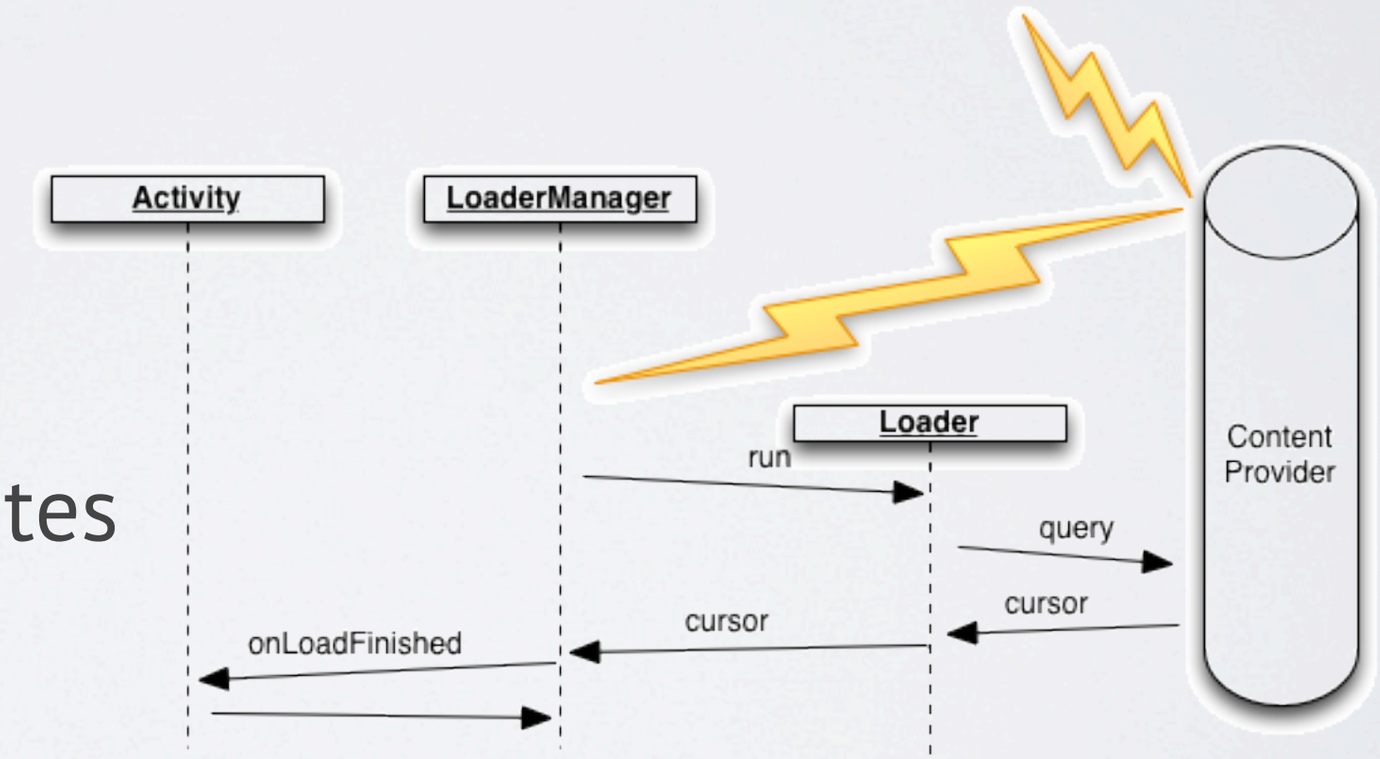
... and runs a new query!

Wait! Say that again?

Yep!

05-22 17:26:24.311 1476-1476/? D/TIMELINE: onCreateLoader
05-22 17:26:24.791 1476-1476/? D/TIMELINE: onLoadFinished
05-22 17:26:25.811 1476-1476/? D/TIMELINE: onLoadFinished
05-22 17:30:06.627 1476-1476/? D/TIMELINE: onLoadFinished

When the dataset changes, the LoaderManager automatically updates the UI!



URI == Dataset

A URI is a virtual name for a dataset

`content://com.twitter/tweets/92`

... a single record

`content://com.twitter/tweets/`

... or an entire collection

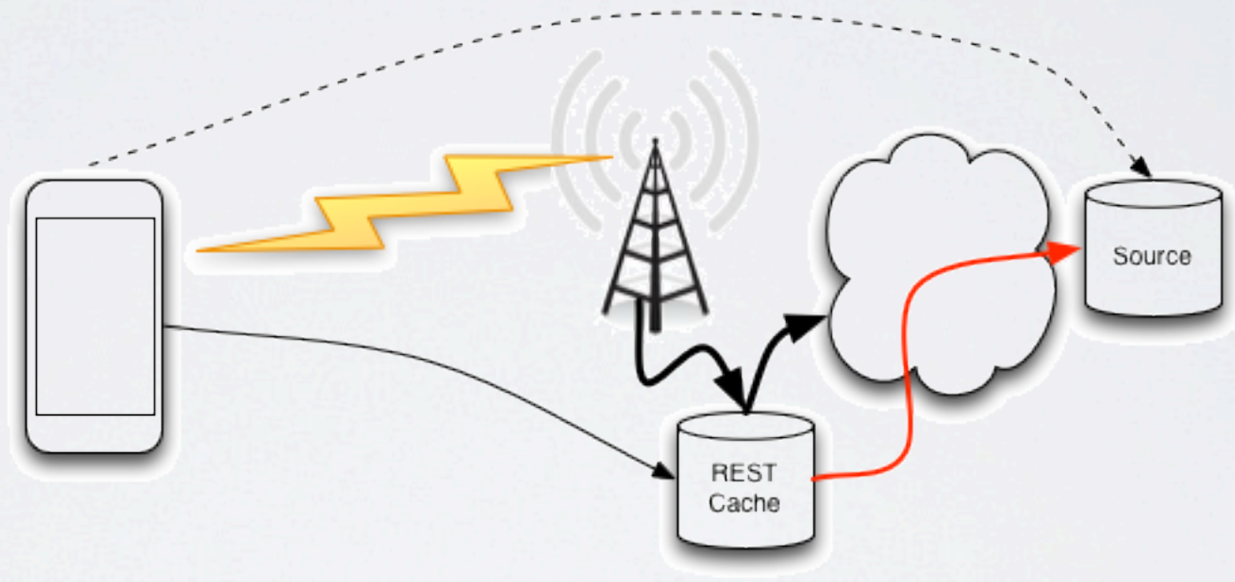
The Content Provider

a Secure, REST abstraction for your data

Your local REST proxy...

REST Scales the Internet

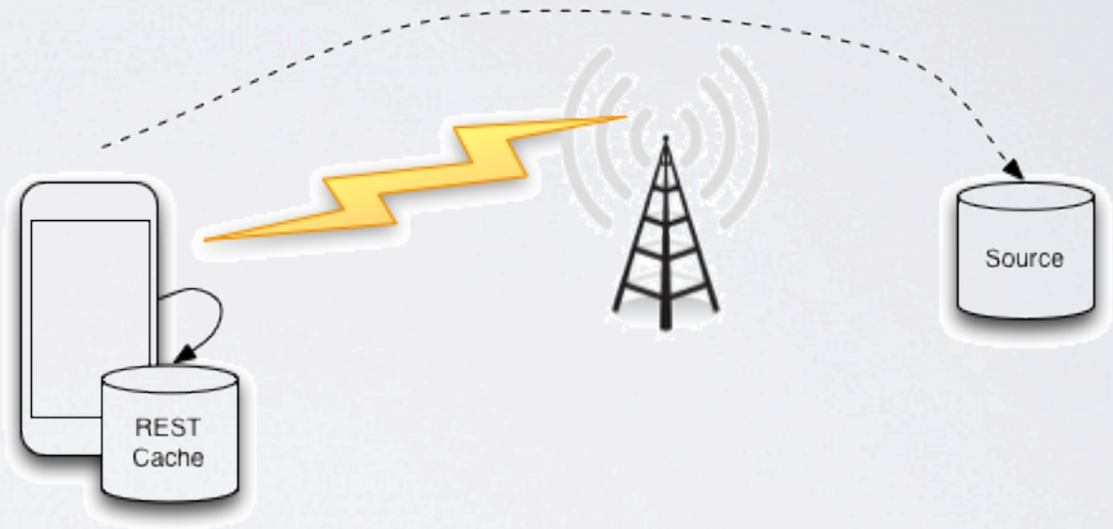
Your phone is connected to a tower whose connection to the Internet is oversold



REST can Scale your App!

The connection between the device and the tower is even:

- less reliable
- more expensive

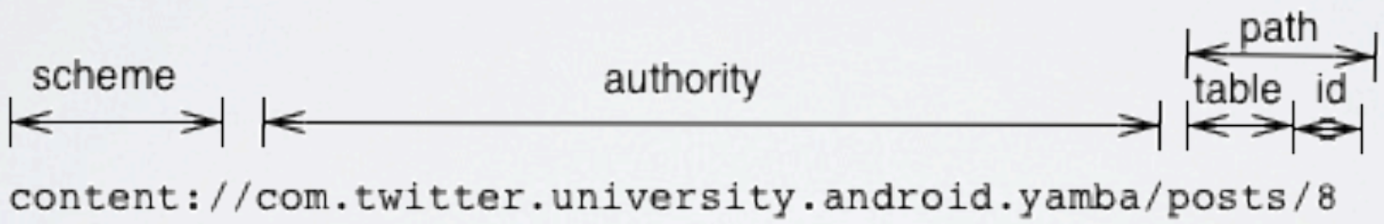


So make the proxy local!

Declaring a Content Provider

The ContentProvider attaches the dataset to its URI

```
<provider
    android:name=".data.YambaProvider"
    android:authorities="com.twitter.university.android.yamba"
    android:readPermission="com.twitter.university.android.yamba.permission.READ"
    android:writePermission="com.twitter.university.android.yamba.permission.WRITE"
/>
```



The Contract

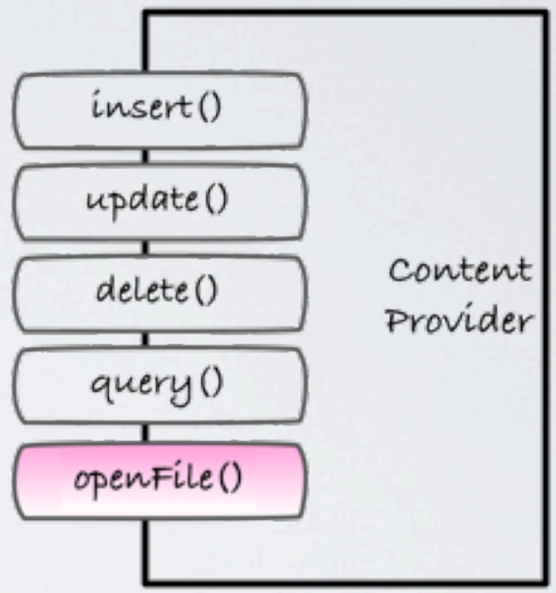
This is your API!

```
public static class Posts {
        private Posts() { }

        public static final String TABLE = "posts";

        public static final Uri URI = BASE_URI.buildUpon().appendPath(TABLE).build();

        private static final String MINOR_TYPE = "/vnd." + AUTHORITY + "." + TABLE;

        public static final String ITEM_TYPE
                = ContentResolver.CURSOR_ITEM_BASE_TYPE + MINOR_TYPE;
        public static final String DIR_TYPE
                = ContentResolver.CURSOR_DIR_BASE_TYPE + MINOR_TYPE;

        public static class Columns {
                private Columns() { }
                public static final String ID = BaseColumns._ID;
                public static final String TIMESTAMP = "timestamp";
                public static final String TRANSACTION = "xact";
                public static final String SENT = "sent";
                public static final String TWEET = "tweet";
        }
}
```

Implementing a Content Provider

Translate REST calls into database queries

```
public Uri insert(Uri uri, ContentValues row) {
    // ...

    long id = getDb().insert(
        YambaDbHelper.TABLE_POSTS,
        null,
        COL_MAP_POSTS.translateCols(row));

    if (0 >= id) { return null; }

    uri = uri.buildUpon().appendPath(String.valueOf(id)).build();
    getContext().getContentResolver().notifyChange(uri, null);

    return uri;
}
```



Virtual Tables

The ContentProvider exposes virtual tables

Consider separating them from physical tables using `SQLiteQueryBuilder`

```
SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
qb.setTables(DBHelper.TABLE);
qb.setProjectionMap(PROJ_MAP);

Cursor c = qb.query(getDb(), proj, sel, selArgs, null, null, sort);
```

The Alias Trick

The QueryBuild `SQLiteQueryBuilder` maps the projection parameter (String[])

SELECT timestamp, tweet
FROM Posts
ORDER BY timestamp

timestamp	=	p_timestamp
tweet	=	p_tweet

SELECT p_timestamp, p_tweet
FROM Posts
ORDER BY timestamp

Fixed with a SQL trick:

SELECT timestamp, tweet
FROM Posts
ORDER BY timestamp

timestamp	=	p_timestamp AS timestamp
tweet	=	p_tweet AS tweet

SELECT p_timestamp AS timestamp, p_tweet AS tweet
FROM Posts
ORDER BY timestamp

Using a Content Provider

CRUD methods, using the ContentResolver

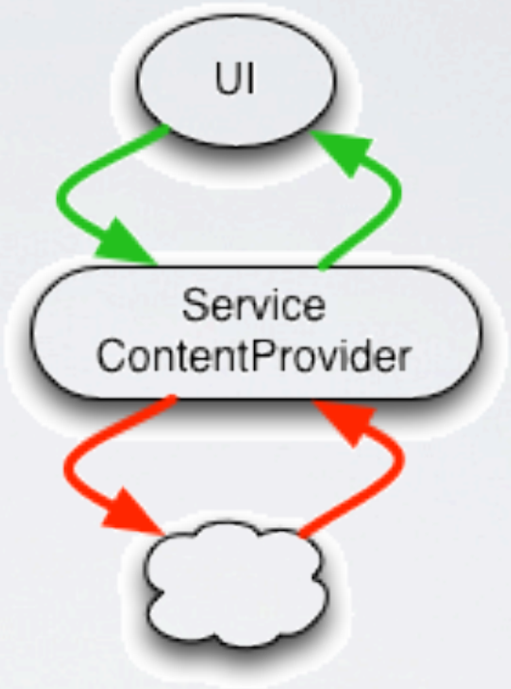
```
Cursor c = ctxt.getContentResolver().query(
    YambaContract.Posts.URI,
    null,
    null,
    null,
    null);
```

```
ContentValues cv = new ContentValues();
cv.put(YambaContract.Posts.Columns.TWEET, tweet);
cv.put(YambaContract.Posts.Columns.TIMESTAMP, now);
ctxt.getContentResolver().insert(YambaContract.Posts.URI, cv);
```

Introducing the SyncAdapter

Network MVC

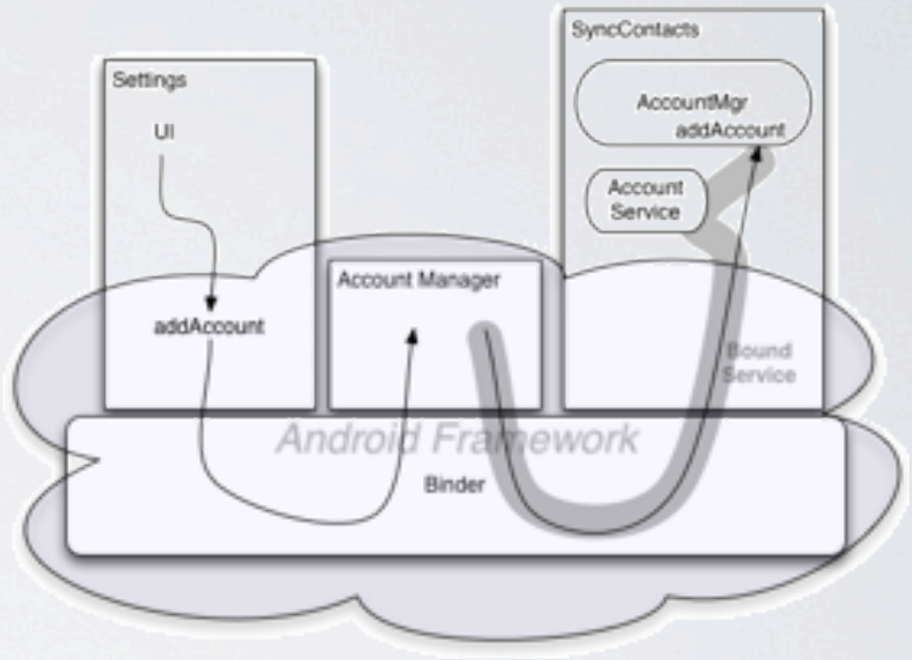
Efficient, Secure,
Poorly Documented

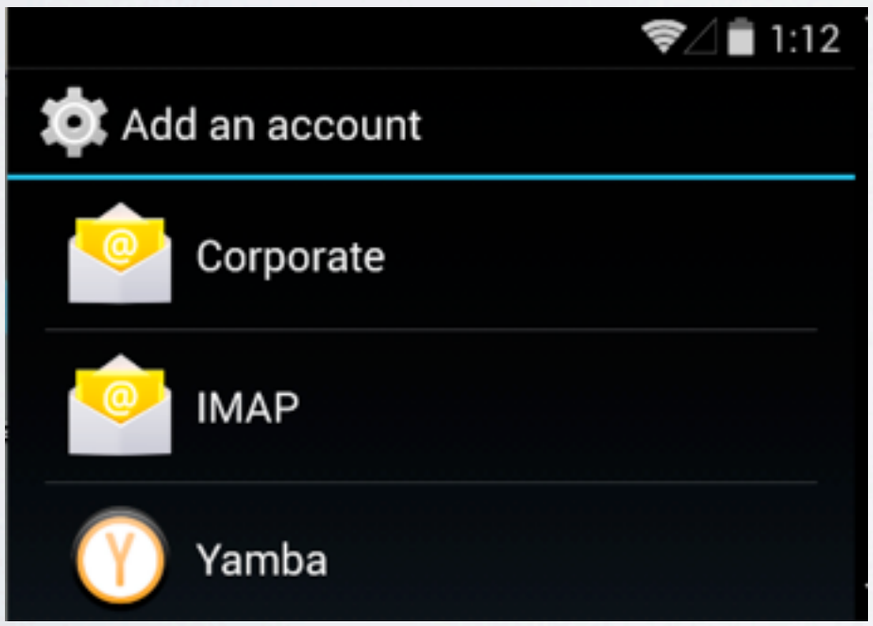


The Account

The unit of synchronization is the Account

An Application whose manifest contains an AccountType definition, is responsible for maintaining accounts of that type.





The Sync Service

A SyncAdapter connects an Account and a URI!

```
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="@string/account_type"
    android:contentAuthority="com.twitter.university.android.yamba"
    android:isAlwaysSyncable="true" />
```

Synchronization

One more bit of ContentProvider magic!

Remember notifying the UI that data had changed?

```
getContext().getContentResolver().notifyChange(uri, null);
```

With a third argument the same call can request network synchronization

```
getContext().getContentResolver().notifyChange(uri, null, true);
```

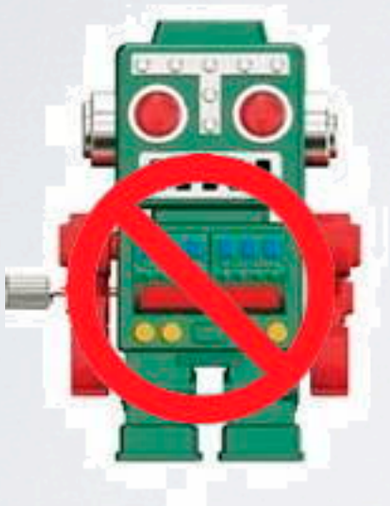
Putting it all together

A Content Provider, in the middle of your application:

- Isolates the UI from the back-end
- Separates data from storage implementation
- Makes your application a re-usable component
- Automates UI and back-end synchronization
- **Provide users with a consistent, reliable view of application state**

No more Async Tasks!

Thanks!



blake.meike@gmail.com

twitter: @callmeike

blog: http://portabledroid.wordpress.com/