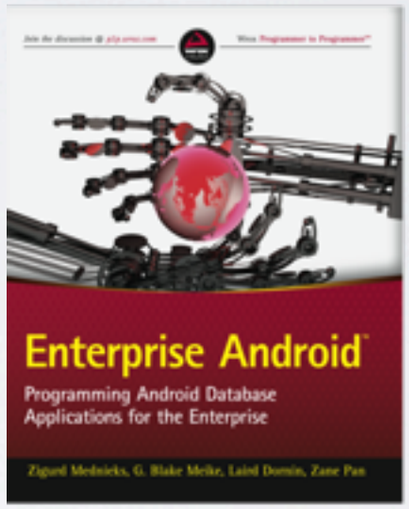
Building Android Apps that aren't Toys

Blake Meike

Developer, Architect, Android Evangelist,

Founding member of Twitter University

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



blake.meike@gmail.com

twitter: @callmeike

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

Schedule

Class runs 9 - 5

- 15 minute breaks at 10:30 and 3:00
- Lunch from 12:30 - 1:30
- All breaks will include labs:
  - Meet Yamba
  - AsyncTask
  - Intent Service
  - Persistent Tweets
  - Dobjanschi architecture
  - Creating an Account I
  - Creating an Account II
  - Creating a Sync Adapter

Lab I: Meet Yamba

Set up your dev env (Java, ADK, Studio)

http://developer.android.com/sdk/installing/bundle.html

- Download Yamba

  https://github.com/bmeike/Yamba.git

  https://github.com/bmeike/EclipseYamba.git

- Install Genymotion?

  https://shop.genymotion.com/index.php?controller=order-opc

- Install and run Yamba
- Attempt to post a tweet

It's broken!

...but you knew that.

So, what's up?

Lab II: Async Task

Please implement an AsyncTask so that the application works:

- Post the tweet on a background thread
- Post a Toast reporting success/failure
- Stretch: Handle mashing the tweet button

Excellent!

What did we accomplish?

Successes

- Network transaction is off the UI Thread

Warning!

- Concurrency Issues
- Lifecycle Issue

Basic Concurrency Issues:

What's the problem here?

```
private static int queryCount;

class QueryTask extends AsyncTask<String, Void, String> {
    @Override
    protected String doInBackground(String... params) {
        queryCount++;
        return RESTHelper.queryServer(params[0]);
    }

    @Override
    protected void onPostExecute(String result) {
        displayPage(result);
    }
}
```

Aliased Concurrency Issues:

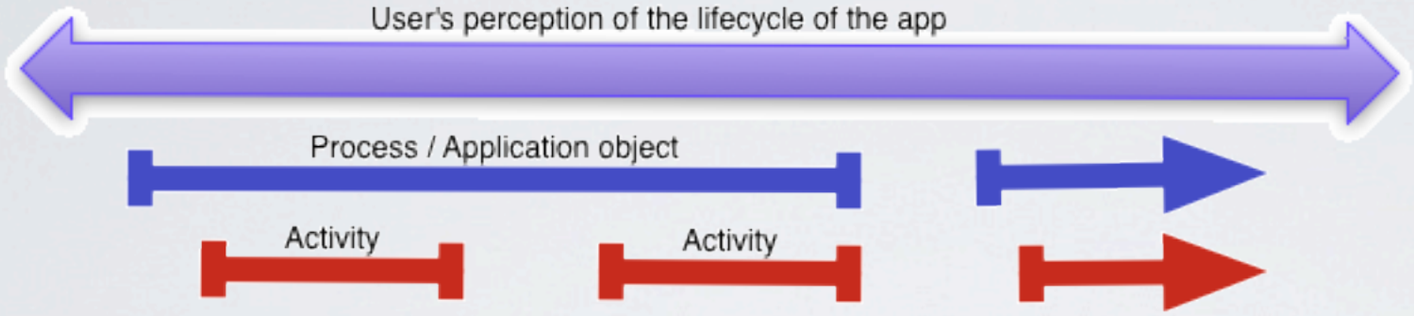
How about here?

```
    private static int queryCount;

class QueryTask extends AsyncTask<List<String>, Void, List<String>> {
    @Override
    protected List<String> doInBackground(List<String>... params) {
        return RESTHelper.queryServer(params[0]);
    }

    @Override
    protected void onPostExecute(List<String> result) {
        queryCount++;
        displayPage(result);
    }
}
```

Android Component Lifecycles:



Basic Lifecycle Issues:

What's wrong with this?

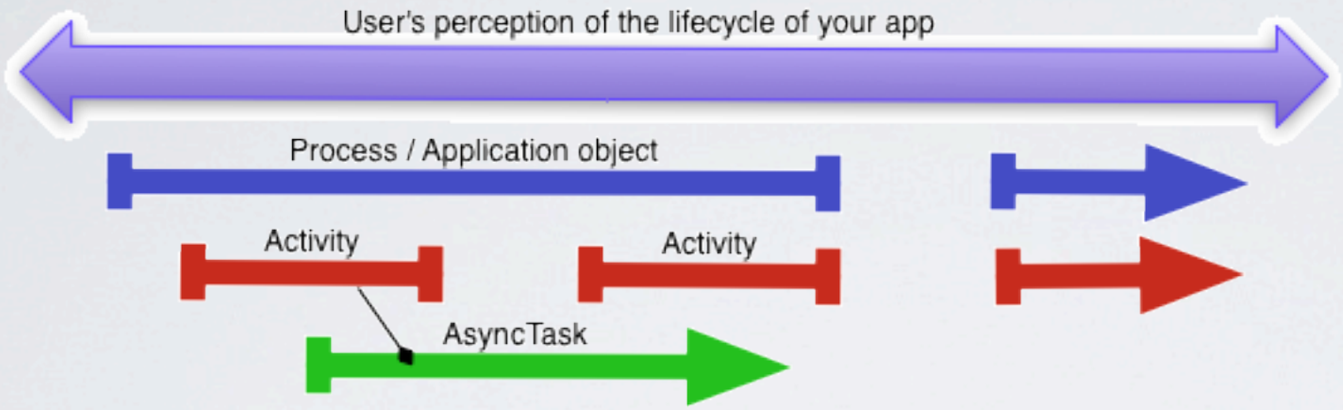
```
static class QueryTask extends AsyncTask<String, Void, String> {
 private final QueryActivityIII activity;

 public QueryTask(QueryActivityIII activity) {
     this.activity = activity;
 }

     @Override
     protected String doInBackground(String... params) {
         return RESTHelper.queryServer(params[0]);
     }

     @Override
     protected void onPostExecute(String result) {
         activity.displayPage(result);
     }
}
```

Android Component Lifecycles:



Scary Lifecycle Issues:

What's wrong with this?

```
class QueryTask extends AsyncTask<String, Void, String> {
 @Override
 protected String doInBackground(String... params) {
     queryCount++;
     return RESTHelper.queryServer(params[0]);
 }

     @Override
     protected void onPostExecute(String result) {
         displayPage(result);
     }
}
```

Hint: it is exactly the same thing

Other issues:

- What happens if you post tweets quickly? Will they be sent in order?
- What happens if you lose connectivity?
- What happens if you lose connectivity and post more tweets?
- What happens if you lose connectivity, post more tweets and start 10 new apps?

Intent Service

Let’s explore a different solution that addresses the Lifecycle issue

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

If Android needs space, it will kill your app!

```
kill -9
```

How does it work?

- Marshall the request into an Intent
- Request posted to a single Looper Thread

Marshalling an Intent:

Static helper method marshals a method call as 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);
}
```

- First param in the method name
- Other params as needed

The UI has been decoupled!

Processing the Intent:

The Intent appears in the `onHandleIntent` method

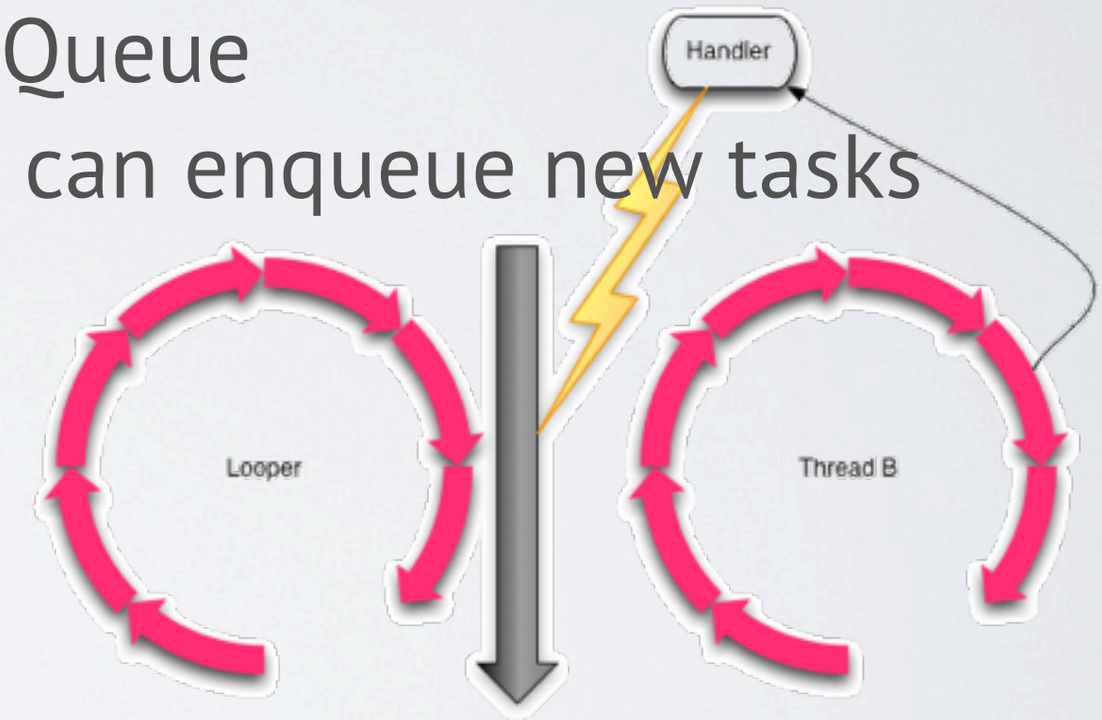
```
protected void onHandleIntent(Intent intent) {
 int op = intent.getIntExtra(PARAM_OP, 0);
 switch(op) {
     case OP_POST:
         String tweet = intent.getIntExtra(PARAM_TWEET, “”);
         if (!TextUtils.isEmpty(tweet) { doPost(tweet); }
         break;

         // ...

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

Looper/Handler

- Basic Android Threading Mechanism
- Three parts:
  - A Queue of tasks
  - A Thread that services the Queue
  - One or more Handlers that can enqueue new tasks



Lab III: Intent Service

Please replace your AsyncTask with an IntentService:

- Call the static helper from the Activity
- Catch the intent in `Service.onHandleIntent`
- Post the tweet from code run on the background thread
- Don’t forget to register your service in the Manifest!
- Stretch: Use a Handler to enqueue at task that will

Excellent!

What did we accomplish?

Successes

- Network transaction is off the UI Thread
- Guaranteed in-order delivery
- Elegant architecture: Activity sees posting as someone else's problem.
- oom_adj is around 4:

Warning!

- Concurrency
- The Handler is awkward

Other issues:

- While it is less likely that a tweet will get lost, it is still not impossible
- Communications with the IntentService are one-way

Bound Service

Let's explore a different solution that addresses the one-way communications problem.

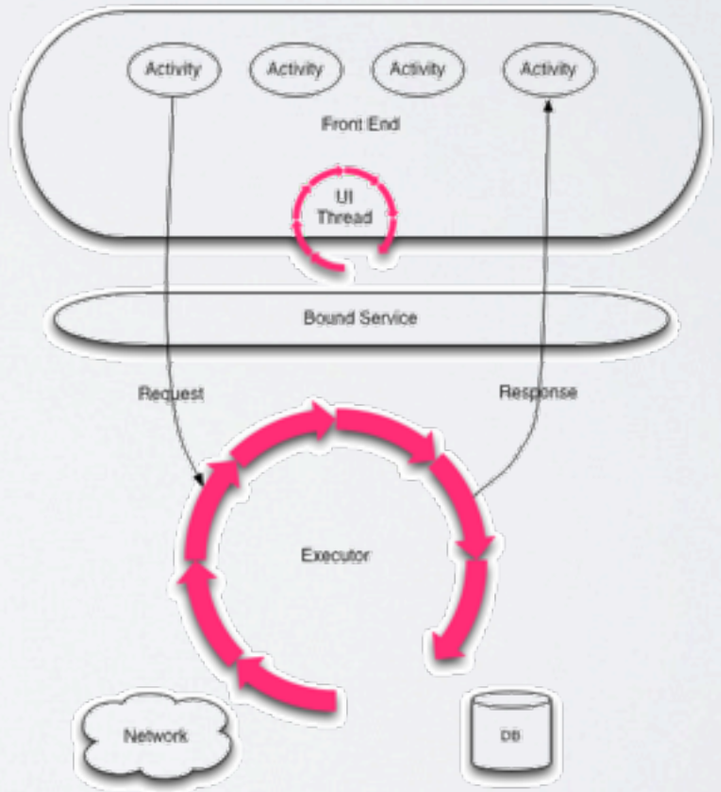
We will not implement this one, but you should be aware of its existence.

Very common architecture for large-scale enterprise apps: Twitter, Facebook, etc

How does it work?

Components

- An Android Service component: Factory for the back-end
- A Service Implementation: Typically a framework for executing template tasks
- TemplateTasks: AsyncTask like task templates



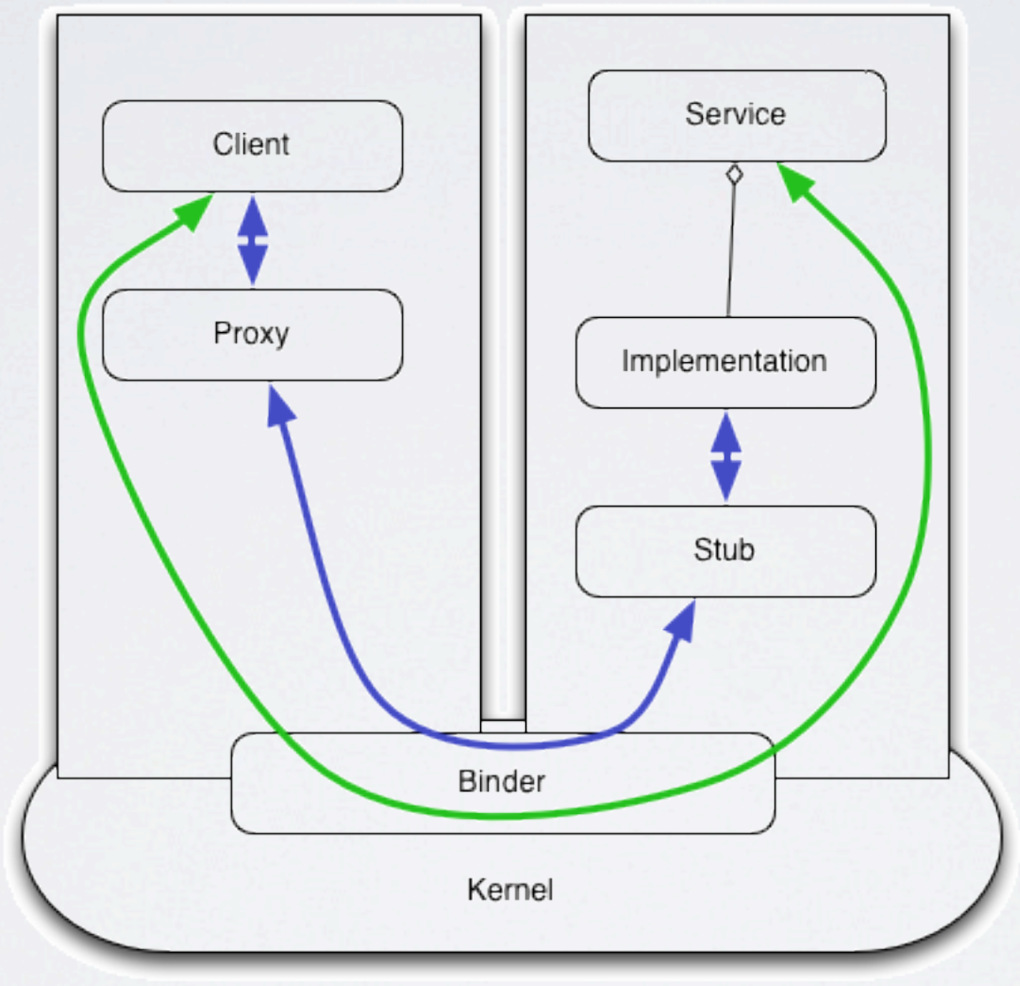
How does it work?

Strategy

Activity binds the Service

- Binding process asynchronously returns an Implementation object
- Activity makes calls on the Implementation object
- API calls are may be executed immediately or submitted to an ExecutionService
- Task submitted to the ExecutionService have Request and Response sections:
  - Request phase is run on the background Executor
  - Response phase is run back on the UI thread

Binding to a Service



Bound Service Code Walkthrough

Excellent!

What did we accomplish?

Successes

- Similar to IntentService
- Rich and flexible, if somewhat complicated, API
- Bound service calls are full-duplex
- Bound service calls are about 2 orders of magnitude faster than Intents

This is ***very*** Web Service.

Warning!

- Need your own ExecutionService
- Need to start and stop the Service yourself
- Activity must handle unbound state!

Other issues:

- Doesn’t explicitly address command persistence

Persistent Commands

Let’s explore yet another solution that makes commands persistent

How does it work?

We'll need this table in the DB

```
CREATE TABLE posts (
 _id INTEGER PRIMARY KEY AUTOINCREMENT,
 timestamp INTEGER NOT NULL,
 xact STRING DEFAULT(NULL),
 sent INTEGER DEFAULT(NULL),
 tweet STRING NOT NULL
);
```

... and this insert statement in the ContentProvider

```
INSERT INTO posts(timestamp, tweet)
   VALUES( <now>, <tweet> );
```

Lab IV: Persistent Tweets

Please modify:

- Contract
  - Add URI and Columns for Post table
- DBHelper
  - add posts table to onCreate
  - add posts table to onUpgrade
  - Version the DB!!
- ContentProvider
  - Add the new table to the Matcher
  - Implement the insert method (return param uri with appended id)
- Logic
  - Create ContentValues with tweet and timestamp
  - Use ContentResolver to insert into DB

```
CREATE TABLE posts (
    _id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp INTEGER NOT NULL,
    xact STRING DEFAULT(NULL),
    sent INTEGER DEFAULT(NULL),
    tweet STRING NOT NULL
);
```

Excellent!

What did we accomplish?

Successes

- Excessively simple
- Tweet is (finally) persistent!

Warning!

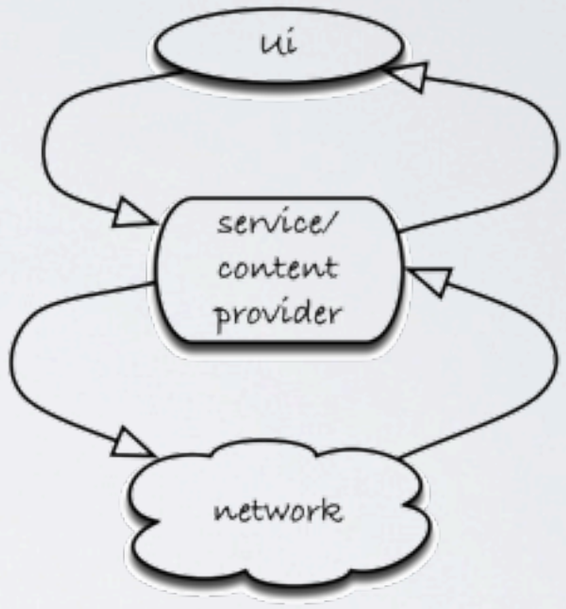
Pretty good, really....

Other issues:

- Tweets aren't being sent!

Dobjanschi Architecture

Let's explore sending tweets from the database



This is the most difficult lab of the day.

How does it work?

Safe Network Transaction

1. Get a UUID
2. Update `xact=UUID` for outbound records with `sent` and `xact==null`
3. Get a cursor full of records with `xact==UUID`
4. Post each record in the cursor to the remote; if post succeeds set `sent=now`
5. To complete the transaction set `xact=null` where `xact==UUID`

Lab IV: Network Transactions

Please modify your IntentService and ContentProvider:

- rename `YambaLogic.doPoll` to `doSync`
  - Call the new method from YambaService
  - Make doPost initiate a sync!
- Implement a network transaction
  - Call the new method from doSync
  - Remember transaction status for each record
- Fix the ContentProvider
  - The `query` method must work for the POSTS table
  - Add an update method

1. Get a UUID
2. Update `xact=UUID` for outbound records with `sent` and `xact==null`
3. Get a cursor full of records with `xact==UUID`
4. Post each record in the cursor to the remote; if post succeeds set `sent=now`
5. To complete the transaction set `xact=null` where `xact==UUID`

Really, really Excellent!

What did we accomplish?

Successes

- Off the UI thread
- Tweets will always get sent, lifecycle be damned!
- Possible to provide UI feedback about sending state

This is ***really*** pretty good

Warning!

- Network CRUD

Other issues:

- crickets.....

Sync Adapter

What we already have is pretty excellent.
We can improve it even more, though.

Consider:

- Using the radio is one of the most battery intensive things the device can do
- Synchronization cannot be asynchronous, if it requires securely stored credentials.

How does it work?

- Define an Account Type
- Create an Account
- Define a SyncAdapter

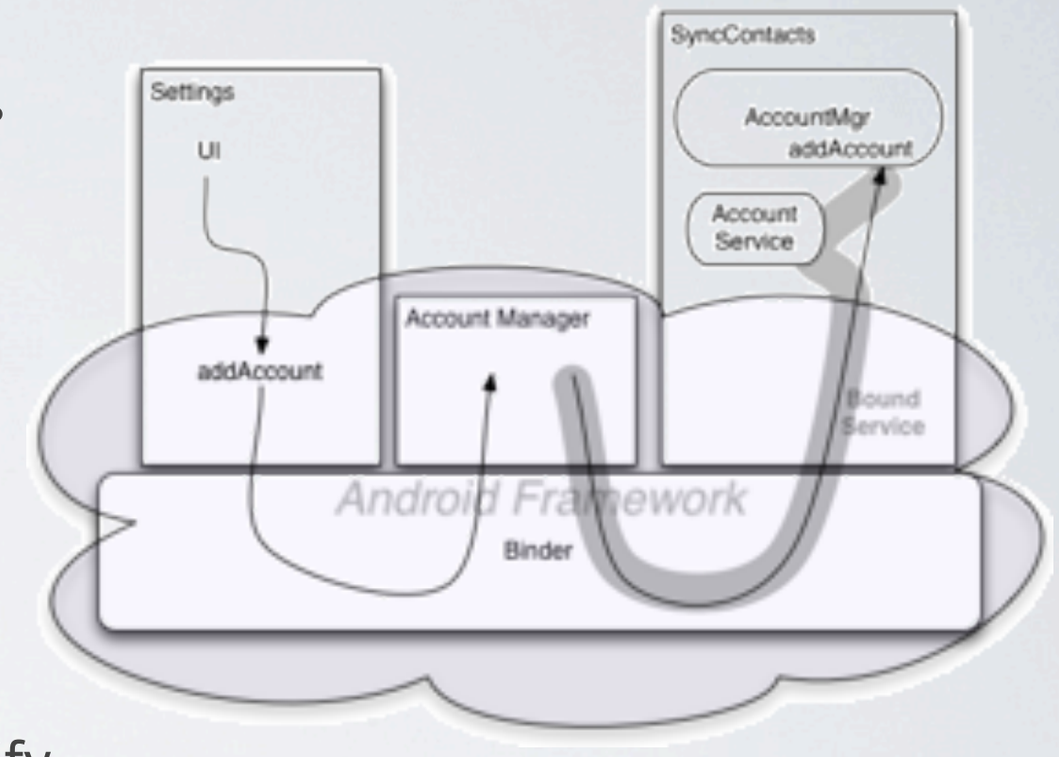
The Account Manager

In order to run a Sync Adapter, we need something for it to synchronize.

In Android, the unit of synchronization, is the Account.

How does it work?

- Declare a service that, on binding, returns an instance of *AbstractAccountAuthenticatorService*
- The declaration must contain a filter for the intent: *android.accounts.AccountAuthenticatorService*
- The declarations must contain a reference to a metadata file
- The reference metadata file must contains a *account-authenticator* element
- The account-authenticator element must specify an *android:accountType*



Lab V: The Account Manager

Add a new Service to your application

- Add to Manifest
  - Declare the Service
  - Add an IntentFilter
  - Write the meta-data file
- Implement the Service
  - Create the service class
  - Return an instance of the manager
- Implement the Manager
  - Subclass AbstractAccountAuthenticator
  - Just log method calls
- Test from System > Accounts

Account Creation

Now that we have an account type, we can create an account of that type

How does it work?

A new account is actually created with the method:

`AccountManager.addAccountExplicitly`

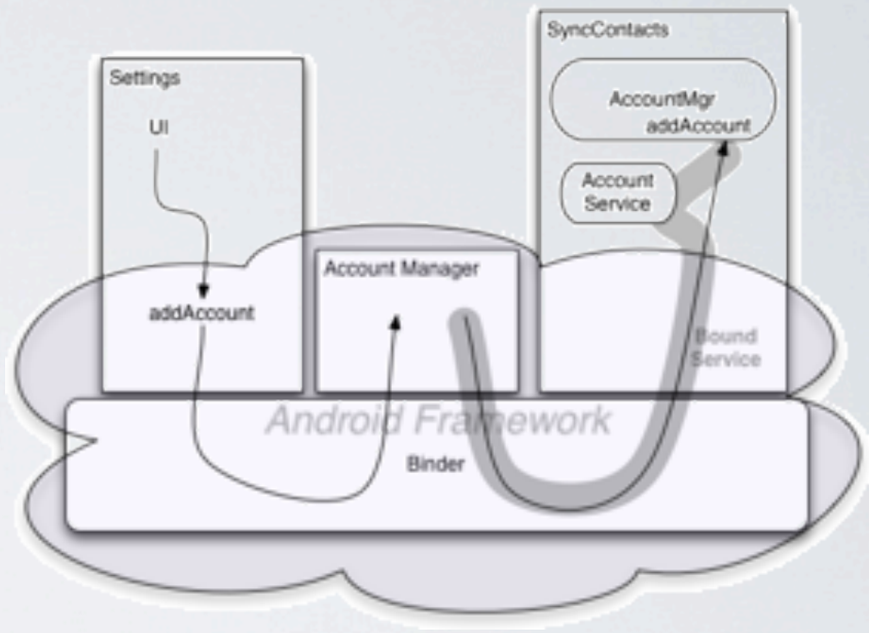
... which, requires the permission:
`android.permission.AUTHENTICATE_ACCOUNTS`

... and probably:
`android.permission.WRITE_SYNC_SETTINGS`
`android.permission.MANAGE_ACCOUNTS`
`android.permission.GET_ACCOUNTS`

How does it work?

Consider the System Settings App

- Requests the AccountManager
- Gets wrapped Binder connection to the AccountManager
- The AccountManager, in turn, binds to the AccountMgrService in your app.
- System Settings calls `addAccount`
  - Returns a Future
  - Call is proxied forward to `addAccount` in your app, passing an AccountAuthenticatorResponse
  - The Future is fulfilled when the `onResponse` is called



How does it work?

Three modes:

- **Immediate**
  `createAccount` returns Bundle that is:
  - non-null
  - does not contain the key: `AccountManager.KeyIntent`

  The future is fulfilled immediately when `createAccount` returns the Bundle. It is assumed that the Bundle contains any necessary information.

- **Intent**
  `createAccount` returns Bundle that is:
  - non-null
  - contains the key: `AccountManager.KeyIntent`

  When `createAccount` returns, Android fires the Intent to start an Activity that will create the account. The future is fulfilled when the Activity calls `finish`

- **Delayed**
  `createAccount` returns Bundle that is:
  - null

  The future is fulfilled only when application code explicitly calls `AccountAuthenticatorResponse.onResult`.

Lab VI: Create an Account

Create a new account in immediate mode by implementing `addAccount`

- Add the account
- Return an appropriate Bundle
- Stretch: Check to see if the account already exists.

SyncAdapter

… and finally, we can synchronize data.

How does it work?

Essentially the same as the AccountService:

- Declare the SyncService
  - Filter for `android.content.SyncAdapter`
  - Refer to meta-data `android.content.SyncAdapter`
- Define meta-data
  - Link to accountType
  - Link to contentAuthority
- Implement the Service: return an instance of a subclass of `AbstractThreadedSyncAdapter`
- Implement `onPerformSync`

How does it work?

Remember:

`ContentResolver.notifyChange(uri, observer)` ??

Well! There is a 3-arg version; the last arg is a boolean.

Guess what it does?

Starts the SyncAdapter!!!

Lab VI: Implement a SyncAdapter

- Define the SyncAdapter Service
  - Intent Filter: android.content.SyncAdapter
  - meta-data: sync-adapter
    - account-type
    - contentAuthority
- Implement AccountMgr.getAuthToken
  - Create a client and store it in the application with its token
  - return the token as AccountManager.KEY_AUTHTOKEN
- Implement the SyncAdapter Service
  - create an subclass of AbstractThreadedSyncAdapter
  - return its getSyncAdapterBinder
  - call getBlockingAuthToken
  - use the returned token to get the client from the Application