CHAPTER 9

The Database

The Android system uses databases to store useful information that needs to be persisted even when the user kills the app or even shuts down the device and powers it back on. The data includes contacts, system settings, bookmarks, and so on.

So, why use a database in a mobile application? After all, isn't it better to keep our data in a cloud where it's always backed up instead of storing it in a mobile device that is easily lost or damaged?

A database in a mobile device is very useful as a supplement to the online world. Although in many cases it is much better to count on the data living in the cloud, it is useful to store it locally in order to access it more quickly and have it available even when the network is not available. In this case, we are using a local database as a cache. This is also how we use it in our Yamba application.

In this chapter, you will learn how Android supports databases. You will learn to create and use a database inside the Yamba application to store our status updates locally. Local data will help Yamba display statuses to the user quickly, without having to wait for the network to provide the data. Our service will run in the background and periodically update the database so that the data is relatively fresh. This will improve the overall user experience of the application.

About SQLite

SQLite is an open source database that has been around for a long time, is quite stable, and is popular on many small devices, including Android. There are couple of good reasons why SQLite is a great fit for Android app development:

- It's a zero-configuration database. That means there's absolutely no database configuration for you as the developer. This makes it relatively simple to use.
- It doesn't have a server. There's no SQLite database process running. It is basically a set of libraries that provide the database functionality. Not having a server to worry about is also a good thing.

- It's a single-file database. This makes database security straightforward, as it boils down to filesystem security. We already know that Android sets aside a special, secure sandbox for each application.
- It's open source.

The Android framework offers several ways to use SQLite easily and effectively, and we'll look at the basic usage in this chapter. You may be pleased to find that, although SQLite uses SQL, Android provides a higher-level library with an interface that is much easier to integrate into an application.

Although SQLite support is built into Android, it is by no means your only option when it comes to data persistence for your app. You can always use another database system, such as JavaDB or MongoDB, but you'd have to bundle the required libraries with your app and would not be able to rely on Android's built-in database support. SQLite is not an alternative to a full SQL server; instead, it is an alternative to using a local file with an arbitrary format.

DbHelper

Android provides an elegant interface for your app to interact with an SQLite database. To access the database, you first need a helper class that provides a "connection" to the database, creating the connection if it doesn't already exist. This class, provided to you by the Android framework, is called `SQLiteOpenHelper`. The database class it returns is an instance of `SQLiteDatabase`.

In the following subsections I'll explain some of the background concepts you should understand when working with `DbHelper`. I'm not going to explain SQL or basic database concepts such as normalization, because there are hundreds of good places to find that information, and I expect most of my readers already know it. However, this chapter should give you enough to get started, even if your knowledge of databases is spotty.

The Database Schema and Its Creation

A schema is just a description of what's in a database. In our Yamba database, for instance, we want fields for the following information about each tweet we retrieve from Twitter:

`created_at`
: The date when the tweet was sent

`txt`
: The text of the tweet

`user`
: The user who sent the tweet

So each row in our table will contain the data for one tweet, and these four items will be the columns in our schema, along with a unique ID for each tweet. We need the ID so we can easily refer to a tweet. SQLite, like most databases, allows us to declare the ID as a primary key and even assigns a unique number automatically to each tweet for us.

The schema has to be created when our application starts, so we'll do it in the `onCreate()` method of DbHelper. We might add new fields or change existing ones in a later version of our application, so we'll assign a version number to our schema and provide an `onUpgrade()` method that we can call to alter the schema.

`onCreate()` and `onUpgrade()` are the only methods in our application when we need to use SQL. We'll execute `CREATE TABLE` in `onCreate()` to create a table in our database. In a production application, we'd use `ALTER TABLE` in `onUpgrade()` when the schema changes, but that requires a lot of complex introspection of the database, so for now we'll use `DROP TABLE` and recreate the table. Of course, `DROP TABLE` destroys any data currently in the table, but that's not a problem for our Yamba application. It always refills the table with tweets from the past 24 hours, which are the only ones our users will care about.

Four Major Operations

The `DbHelper` class offers you a high-level interface that's much simpler than SQL. The developers realized that most applications use databases for only four major operations, which go by the appealing acronym CRUD: create, read (query), update, and delete. To fulfill these requirements, `DbHelper` offers:

`insert()`
: Inserts one or more rows into the database

`query()`
: Requests rows matching the criteria you specify

`update()`
: Replaces ones or more rows that match the criteria you specify

`delete()`
: Deletes rows matching the criteria you specify

Each of these methods has variants that enhance it with other functions. To use one of the methods, create a `ContentValues` container and place in it the information you want inserted, updated, etc. This chapter will show you the process for an insert, and the other operations work in similar ways.

So, why not use SQL directly? There are three good reasons why.

First, from a security point of view, an SQL statement is a prime candidate for a security attack on your application and data, known as an SQL injection attack. That is because the SQL statement takes user input, and unless you check and isolate it very carefully, this input could embed other SQL statements with undesirable effects.

Second, from a performance point of view, executing SQL statements repeatedly is highly inefficient because you'd have to parse the SQL every time the statement runs.

Finally, the `DbHelper` methods are more robust and less likely to pass through the compiler with undetected errors. When you include SQL in a program, it's easy to create errors that turn up only at runtime.

Android's database framework only supports prepared statements for standard CRUD operations: INSERT, UPDATE, DELETE, and SELECT. For other SQL statements, we pass them directly to SQLite. That's why we used `execSQL()` to run the code to `CREATE TABLE...`. This is OK because that code doesn't depend on any user input, and as such SQL injection is not possible. Additionally, that code runs very rarely, so there's no need to worry about the performance implications.

Cursors

A query returns a set of rows along with a pointer called a *cursor*. You can retrieve results one at a time from the cursor, causing it to advance each time to the next row. You can also move the cursor around in the result set. An empty cursor indicates that you've retrieved all the rows.

In general, anything you do with SQL could lead to an SQL exception because it's code is interacting with a system that's outside of our direct control. For example, the database could be running out of space or somehow corrupted. So, it is a good practice to handle all the `SQLException`s by surrounding your database calls in try/catch blocks.

It's easy to do this using the Eclipse shortcut:

1. Select the code for which you'd like to handle exceptions. Typically this would be most of your SQL calls.
2. In the Eclipse menu, choose Source→Surround With→Try/catch Block. Eclipse will generate the appropriate try/catch statements around your code for the proper exception class.
3. Handle this exception in the `catch` block. This might be a simple call to `Log.e()` to pass the tag, message, and the exception object itself.

First Example

So now we're going to create our own helper class to help us open our Yamba database (see Example 9-1). We'll call the class `DbHelper`. It will create the database file if one

doesn't already exist, or it will upgrade the user's database if the schema has changed between versions.

Like many other classes in Android, we usually start by subclassing a framework class, in this case `SQLiteOpenHelper`. We then need to implement the class's constructor, as well as `onCreate()` and `onUpgrade()` methods.

Example 9-1. DbHelper.java, version 1

```
package com.marakana.yamba4;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.provider.BaseColumns;
import android.util.Log;

public class DbHelper1 extends SQLiteOpenHelper { // ❶
  static final String TAG = "DbHelper";
  static final String DB_NAME = "timeline.db"; // ❷
  static final int DB_VERSION = 1; // ❸
  static final String TABLE = "timeline"; // ❹
  static final String C_ID = BaseColumns._ID;
  static final String C_CREATED_AT = "created_at";
  static final String C_SOURCE = "source";
  static final String C_TEXT = "txt";
  static final String C_USER = "user";
  Context context;

  // Constructor
  public DbHelper1(Context context) { // ❺
    super(context, DB_NAME, null, DB_VERSION);
    this.context = context;
  }

  // Called only once, first time the DB is created
  @Override
  public void onCreate(SQLiteDatabase db) {
    String sql = "create table " + TABLE + " (" + C_ID + " int primary key, "
    + C_CREATED_AT + " int, " + C_USER + " text, " + C_TEXT + " text)"; // ❻

    db.execSQL(sql);  // ❼

    Log.d(TAG, "onCreated sql: " + sql);
  }

  // Called whenever newVersion != oldVersion
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) { // ❽
    // Typically do ALTER TABLE statements, but...we're just in development,
    // so:

    db.execSQL("drop table if exists " + TABLE); // drops the old database
    Log.d(TAG, "onUpdated");
    onCreate(db); // run onCreate to get new database
```

```
  }
}
```

❶ Start by subclassing `SQLiteOpenHelper`.

❷ This is the database filename.

❸ This is the version of our database. The version number is important so that later, when you change the schema, you can provide existing users with a way to upgrade their database to the latest schema.

❹ The following are some database constants specific to our application. It is handy to define these as constants to that we can refer to them from other classes.

❺ We override `SQLiteOpenHelper` by passing the constants to `super` and retaining the local reference to the context.

❻ This is the actual SQL that we'll pass on to the database to have it create the appropriate SQL schema that we need.

❼ Once we have our SQL to create the database, run `execSQL ()` on the database object that was passed into `onCreate()`.

❽ `onUpgrade()` is called whenever the user's database version is different than the application version. This typically happens when you change the schema and release the application update to users who already have older version of your app.

As mentioned earlier, you would typically execute `ALTER TABLE ...` SQL statements in `onUpgrade()`. Since we don't have an old database to alter, we are assuming this application is still in prerelease mode and are just deleting any user data when recreating the database.

Next, we need to update the service in order to have it open up the database connection, fetch the data from the network, and insert it into the database.

Update UpdaterService

Remember that our `UpdaterService` connects to the cloud and gets the data. So `UpdaterService` also is responsible for inserting this data into the local database.

In Example 9-2, we update the `UpdaterService` to pull the data from the cloud and store it in the database.

Example 9-2. UpdaterService.java, version 1

```
package com.marakana.yamba4;

import java.util.List;
```

```
import winterwell.jtwitter.Twitter;
import winterwell.jtwitter.TwitterException;
import android.app.Service;
import android.content.ContentValues;
import android.content.Intent;
import android.database.sqlite.SQLiteDatabase;
import android.os.IBinder;
import android.util.Log;

public class UpdaterService1 extends Service {
  private static final String TAG = "UpdaterService";

  static final int DELAY = 60000; // wait a minute
  private boolean runFlag = false;
  private Updater updater;
  private YambaApplication yamba;

  DbHelper1 dbHelper; // ❶
  SQLiteDatabase db;

  @Override
  public IBinder onBind(Intent intent) {
    return null;
  }

  @Override
  public void onCreate() {
    super.onCreate();
    this.yamba = (YambaApplication) getApplication();
    this.updater = new Updater();

    dbHelper = new DbHelper1(this); // ❷

    Log.d(TAG, "onCreated");
  }

  @Override
  public int onStartCommand(Intent intent, int flag, int startId) {
    if (!runFlag) {
      this.runFlag = true;
      this.updater.start();
      ((YambaApplication) super.getApplication()).setServiceRunning(true);

      Log.d(TAG, "onStarted");
    }
    return Service.START_STICKY;
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    this.runFlag = false;
    this.updater.interrupt();
    this.updater = null;
```

```
    this.yamba.setServiceRunning(false);

    Log.d(TAG, "onDestroyed");
  }

  /**
   * Thread that performs the actual update from the online service
   */
  private class Updater extends Thread {
    List<Twitter.Status> timeline;

    public Updater() {
      super("UpdaterService-Updater");
    }

    @Override
    public void run() {
      UpdaterService1 updaterService = UpdaterService1.this;
      while (updaterService.runFlag) {
        Log.d(TAG, "Updater running");
        try {
          // Get the timeline from the cloud
          try {
            timeline = yamba.getTwitter().getFriendsTimeline(); // ❸
          } catch (TwitterException e) {
            Log.e(TAG, "Failed to connect to twitter service", e);
          }

          // Open the database for writing
          db = dbHelper.getWritableDatabase(); // ❹

          // Loop over the timeline and print it out
          ContentValues values = new ContentValues(); // ❺
          for (Twitter.Status status : timeline) { // ❻
            // Insert into database
            values.clear(); // ❼
            values.put(DbHelper1.C_ID, status.id);
            values.put(DbHelper1.C_CREATED_AT, status.createdAt.getTime());
            values.put(DbHelper1.C_SOURCE, status.source);
            values.put(DbHelper1.C_TEXT, status.text);
            values.put(DbHelper1.C_USER, status.user.name);
            db.insertOrThrow(DbHelper1.TABLE, null, values); // ❽

            Log.d(TAG, String.format("%s: %s", status.user.name, status.text));
          }

          // Close the database
          db.close(); // ❾

          Log.d(TAG, "Updater ran");
          Thread.sleep(DELAY);
        } catch (InterruptedException e) {
          updaterService.runFlag = false;
        }
      }
```

```
    }
  } // Updater

}
```

❶ Because we likely need `db` and `dbHelper` objects throughout the class, we declare them globally to the class.

❷ Create the instance of `DbHelper` and pass `this` as its context. This works because the Android `Service` class is a subclass of `Context`. `DbHelper` will figure out whether the database needs to be created or upgraded.

❸ We need to connect to the online service, get the latest updates, and insert them into the database. `getTwitter()` in `YambaApplication` is our lazy initialization of the `Twitter` object. Then, we call the actual Twitter API call `getFriendsTimeline()` to get the last 20 statuses from friends posted in the last 24 hours.

❹ Get the writable database so we can insert new statuses into it. The first time we make this call, `onCreate()` in `DbHelper` will run and create the database file for this user.

❺ `ContentValues` is a simple name-value pair data structure that maps database table names to their respective values.

❻ We loop over all the status data that we received. In this case, we are using a Java for-each loop to make the iteration simple.

❼ For each record, we create a content value. We are reusing the same Java object, clearing it each time we start the loop and populating appropriate values for the status data.

❽ We insert the content value into the database via an `insert()` call to the `SQLiteDatabase` object. Notice that we are not piecing together an SQL statement here, but rather using a *prepared statement* approach to inserting into the database.

❾ Finally, remember to close the database. This is important because another activity could be trying to read or write from this shared resource.

We are now ready to run our code and test it to make sure everything works.

Testing the Service

At this point, we can test whether the database was created properly and whether the service has populated it with some data. We're going to do this step by step.

Verify that the database was created

If the database file was created successfully, it will be located in the */data/data/com.marakana.yamba/databases/timeline.db* file. You can use the Eclipse DDMS perspective and File Explorer view to look at the filesystem of the device, or you can use

`adb shell` on your command line, and then run `ls /data/data/com.marakana.yamba/databases/timeline.db` to make sure the file is there.

To use File Explorer in Eclipse, either open the DDMS perspective in the top-right corner of your Eclipse or go to Windows→Show View→Other...→Android→File Explorer. This will open the view of the filesystem of the device you are currently looking at.

So far, you know that the database file is there, but don't really know whether the database schema was created properly. The next section addresses that.

Using sqlite3

Android ships with the command-line tool `sqlite3`. This tool gives you access to the database itself.

To see whether your database schema was created properly:

1. Open up your terminal or command-line window.
2. Type `adb shell` to connect to your running emulator or physical phone.
3. Change the directory to the location of your database file by typing `cd /data/data/com.marakana.yamba/databases/`.
4. Connect to the database with the `sqlite3 timeline.db` command.

At this point, you should be connected to the database. Your prompt should be `sqlite>`, indicating that you are inside the SQLite:

```
[user:~]> adb shell
# cd /data/data/com.marakana.yamba/databases/
# ls
timeline.db
# sqlite3 timeline.db
SQLite version 3.6.22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

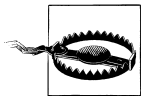
At this point, you can send two types of commands to your SQLite database:

- Standard SQL commands, such as `insert ...`, `update ...`, `delete ...`, and `select ...`, as well as `create table ...`, `alter table ...`, and so on. Note that SQL is another language altogether, and as such is not covered in this book. We assume here that you have a very basic knowledge of SQL. Also note that in `sqlite3`, you must terminate your SQL statements with a semi-colon (`;`).
- `sqlite3` commands. These are commands that are specific to SQLite. You can see the list of all commands by typing `.help` at the `sqlite3>` prompt. For now, we'll just use `.schema` to verify that the schema was created:

```
# sqlite3 timeline.db
SQLite version 3.6.22
```

```
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .schema
CREATE TABLE android_metadata (locale TEXT);
CREATE TABLE timeline ( _id integer primary key,created_at integer, source text,
       txt text, user text );
```

The last line tells us that our database table `timeline` indeed was created and looks like we expected, with the columns `_id`, `created_at`, `source`, `txt`, and `user`.



New Android developers often execute the `sqlite3 timeline.db` command in a wrong folder, and then wonder why the database table wasn't created. SQLite will not complain if the file you are referring to doesn't exist; it will simply create a brand-new database. So, make sure you are either in the correct folder (*/data/data/com.marakana.yamba/databases/*) when you execute `sqlite3 timeline.db`, or run the command specifying the full path to your file: `sqlite3 /data/data/com.marakana.yamba/databases/timeline.db`.

Now that we have a way to create and open up our database, we are ready to update the service that will insert the data into the database.

At this point we should be getting the data from the online service as well as inserting that data in the database. We can also verify that the data is indeed in the database by using `sqlite3`.

Database Constraints

When your service runs for the second time, you'll notice that it fails and that you get many `SQLExceptions` in the LogCat. You will also notice that it complains about the database constraint failing.

This happens because we have duplicate IDs. If you remember, we are fetching all the data from the online service, including the IDs used online. We are then inserting this in to our local database. But we get the data via the `getFriendsTimeline()` call, which returns the 20 most recent posts made in the past 24 hours, and we do this every minute or so. So, unless you have friends who post more than 20 posts a minute, you'll likely get duplicates. That means we're attempting to insert duplicate IDs into a database that is set up with `_id` as the primary key, which means they must be unique. This fails for duplicate entries, and that's why the database complains and throws an `SQLException`.

We could check with the database that there are no duplicates before performing an insert, but that would mean writing that logic. Since the database is already good at database stuff, it is more efficient to attempt to insert duplicate entries, fail at it, and ignore that failure.

To do that, we need to change `db.insert()` to `db.insertOrThrow()`, catch the `SQLException`, and ignore it:

```
...
try {
  db.insertOrThrow(DbHelper.TABLE, null, values);  // ❶
  Log.d(TAG, String.format("%s: %s", status.user.name, status.text));
} catch (SQLException e) {  // ❷
  // Ignore exception
}
...
```

❶ Attempts to insert into the database, but if it fails, it throws an exception.

❷ We catch this exception and ignore it. We will improve on this later in the next section.

At this point, our code works, but it's not ideal. There's an opportunity to refactor it further.

Refactoring Status Data

The work we did previously for the `UpdaterService` is not ideal for supporting our next user of this data: the `TimelineActivity`. Since `TimelineActivity` will also need to access the same database and fetch the same data, it would be better if we would share some of the same functionality between the `UpdaterService` and the `TimelineActivity`.

In order to do that, we'll create a new Java class, `StatusData`, and make it the common container for database-related functionality (see Example 9-3). It will be hiding (encapsulating) SQLite in a higher-level class accessible to other parts of the Yamba application. The rest of our app will then just ask for `StatusData` and will not be concerned with how that data is generated. This is a better design and later will allow us to improve it even further with Content Providers, as explained in Chapter 12.

Example 9-3. StatusData.java

```
package com.marakana.yamba4;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

public class StatusData { // ❶
  private static final String TAG = StatusData.class.getSimpleName();

  static final int VERSION = 1;
  static final String DATABASE = "timeline.db";
  static final String TABLE = "timeline";

  public static final String C_ID = "_id";
  public static final String C_CREATED_AT = "created_at";
  public static final String C_TEXT = "txt";
```

```
public static final String C_USER = "user";

private static final String GET_ALL_ORDER_BY = C_CREATED_AT + " DESC";

private static final String[] MAX_CREATED_AT_COLUMNS = { "max("
    + StatusData.C_CREATED_AT + ")" };

private static final String[] DB_TEXT_COLUMNS = { C_TEXT };

// DbHelper implementations
class DbHelper extends SQLiteOpenHelper {

  public DbHelper(Context context) {
    super(context, DATABASE, null, VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    Log.i(TAG, "Creating database: " + DATABASE);
    db.execSQL("create table " + TABLE + " (" + C_ID + " int primary key, "
        + C_CREATED_AT + " int, " + C_USER + " text, " + C_TEXT + " text)");
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    db.execSQL("drop table " + TABLE);
    this.onCreate(db);
  }
}

private final DbHelper dbHelper; // ❷

public StatusData(Context context) {  // ❸
  this.dbHelper = new DbHelper(context);
  Log.i(TAG, "Initialized data");
}

public void close() { // ❹
  this.dbHelper.close();
}

public void insertOrIgnore(ContentValues values) {  // ❺
  Log.d(TAG, "insertOrIgnore on " + values);
  SQLiteDatabase db = this.dbHelper.getWritableDatabase();  // ❻
  try {
    db.insertWithOnConflict(TABLE, null, values,
        SQLiteDatabase.CONFLICT_IGNORE);  // ❼
  } finally {
    db.close(); // ❽
  }
}

/**
 *
 * @return Cursor where the columns are _id, created_at, user, txt
```

```
     */
    public Cursor getStatusUpdates() {  // ❾
      SQLiteDatabase db = this.dbHelper.getReadableDatabase();
      return db.query(TABLE, null, null, null, null, null, GET_ALL_ORDER_BY);
    }

    /**
     *
     * @return Timestamp of the latest status we ahve it the database
     */
    public long getLatestStatusCreatedAtTime() {  // ❿
      SQLiteDatabase db = this.dbHelper.getReadableDatabase();
      try {
        Cursor cursor = db.query(TABLE, MAX_CREATED_AT_COLUMNS, null, null, null,
            null, null);
        try {
          return cursor.moveToNext() ? cursor.getLong(0) : Long.MIN_VALUE;
        } finally {
          cursor.close();
        }
      } finally {
        db.close();
      }
    }

    /**
     *
     * @param id of the status we are looking for
     * @return Text of the status
     */
    public String getStatusTextById(long id) {  // ⓫
      SQLiteDatabase db = this.dbHelper.getReadableDatabase();
      try {
        Cursor cursor = db.query(TABLE, DB_TEXT_COLUMNS, C_ID + "=" + id, null,
            null, null, null);
        try {
          return cursor.moveToNext() ? cursor.getString(0) : null;
        } finally {
          cursor.close();
        }
      } finally {
        db.close();
      }
    }


}
```

❶ Most of the `StatusData` code is a direct cut-and-paste from *DbHelper.java*. This is because it now makes sense to make `DbHelper` an inner class because `DbHelper` now exists only in the context of `StatusData` and is private to it. In other words, outside of `StatusData`, no other part of the system is concerned with the fact that we are using a database to store our data. That also makes our system flexible, which we will see later with the use of Content Providers.

❷ This is the private and final reference to the `dbHelper` instance. Making it `final` ensures that this object is created only once, whichever part of the system requests it first.

❸ The constructor simply constructs a new instance of `DbHelper`.

❹ We need to expose `close()` for the `dbHelper` so users of it close it properly.

❺ This is the new and improved version of the `db.insert...()` method that we had in `DbHelper` previously.

❻ We open the database only when we need it, which is right before writing to it.

❼ In this case, we use `insertWithOnConflict()` and pass `SQLiteDatabase.CONFLICT_IGNORE` as the final parameter to indicate that when there's a conflict, the exception should be ignored. Remember that we did have a conflict with the duplicate IDs, as explained in "Database Constraints" on page 129.

❽ Notice that we close the database right after we are done. We do this in the `finally` section of our exception handling. This ensures the database is shut down properly, regardless of whether something went wrong. This theme is something we repeat in `getLatestStatusCreatedAtTime()` and `getStatusTextById()`.

❾ This method simply returns all the statuses in the database, with the latest first.

❿ `getLatestStatusCreatedAtTime()` returns the timestamp of the latest status in the database. Having a way to determine the newest locally cached status is useful later, to ensure we add only new statuses into the database.

⓫ For a given ID, `getStatusTextById()` returns the actual text of this status.

Now that we have a new common place to handle status data, we can have it hang off of our common `Application` object so that any part of the application can access the data easily (see Example 9-4). Consequently, the `UpdaterService` and `TimelineActivity` classes are in a has-a relationship to `StatusData` via the `YambaApplication` object.

Example 9-4. YambaApplication.java

```
...
private StatusData statusData; // ❶
...

public StatusData getStatusData() { // ❷
  return statusData;
}

// Connects to the online service and puts the latest statuses into DB.
// Returns the count of new statuses
public synchronized int fetchStatusUpdates() {  // ❸
  Log.d(TAG, "Fetching status updates");
  Twitter twitter = this.getTwitter();
```

```
  if (twitter == null) {
    Log.d(TAG, "Twitter connection info not initialized");
    return 0;
  }
  try {
    List<Status> statusUpdates = twitter.getFriendsTimeline();
    long latestStatusCreatedAtTime = this.getStatusData()
        .getLatestStatusCreatedAtTime();
    int count = 0;
    ContentValues values = new ContentValues();
    for (Status status : statusUpdates) {
      values.put(StatusData.C_ID, status.getId());
      long createdAt = status.getCreatedAt().getTime();
      values.put(StatusData.C_CREATED_AT, createdAt);
      values.put(StatusData.C_TEXT, status.getText());
      values.put(StatusData.C_USER, status.getUser().getName());
      Log.d(TAG, "Got update with id " + status.getId() + ". Saving");
      this.getStatusData().insertOrIgnore(values);
      if (latestStatusCreatedAtTime < createdAt) {
        count++;
      }
    }
    Log.d(TAG, count > 0 ? "Got " + count + " status updates"
        : "No new status updates");
    return count;
  } catch (RuntimeException e) {
    Log.e(TAG, "Failed to fetch status updates", e);
    return 0;
  }
}

...
```

❶ The Yamba application now encapsulates the status data as a private `StatusData` object.

❷ This object is available to the rest of the application for viewing only via this accessor method.

❸ This is where we moved most of the code from the previous version of the `UpdaterService`. This was the code that was running on the `Updater` thread, connecting to the online service to get the data, and then saving that data in the database.

We can now simplify the `UpdaterService` so it uses the refactored code in the `YambaApplication` to get the latest data (see Example 9-5). Note that most of the `Updater`'s `run()` method has been moved to `YambaApplication`'s `fetchStatusUpdates()` method. In addition, the `Updater` doesn't need any access to the `StatusData` object, which is totally hidden from it.

Example 9-5. UpdaterService.java

```
...

private class Updater extends Thread {
```

```
  public Updater() {
    super("UpdaterService-Updater");
  }

  @Override
  public void run() {
    UpdaterService updaterService = UpdaterService.this;
    while (updaterService.runFlag) {
      Log.d(TAG, "Running background thread");
      try {
        YambaApplication yamba = (YambaApplication) updaterService
            .getApplication();  // ❶
        int newUpdates = yamba.fetchStatusUpdates(); // ❷
        if (newUpdates > 0) { // ❸
          Log.d(TAG, "We have a new status");
        }
        Thread.sleep(DELAY);
      } catch (InterruptedException e) {
        updaterService.runFlag = false;
      }
    }
  }
} // Updater

...
```

❶ We get the reference to the `YambaApplication` object, which is readily available to the Android `Service` and thus our own `UpdaterService` instance.

❷ We use the newly created `fetchStatusUpdates()` method in `YambaApplication`, which now houses most of the functionality that was previously part of this `run()` method.

❸ One feature of `fetchStatusUpdates()` is that it returns the number of new records that were fetched. We can use this info for debugging for now, but later will use it for an additional purpose.

Summary

At this point, Yamba can pull the statuses of our friends from the cloud and post them into the local database. We still don't have a way to view this data, but we can verify that the data is there in the database.

Figure 9-1 illustrates what we have done so far as part of the design outlined earlier in Figure 5-4.

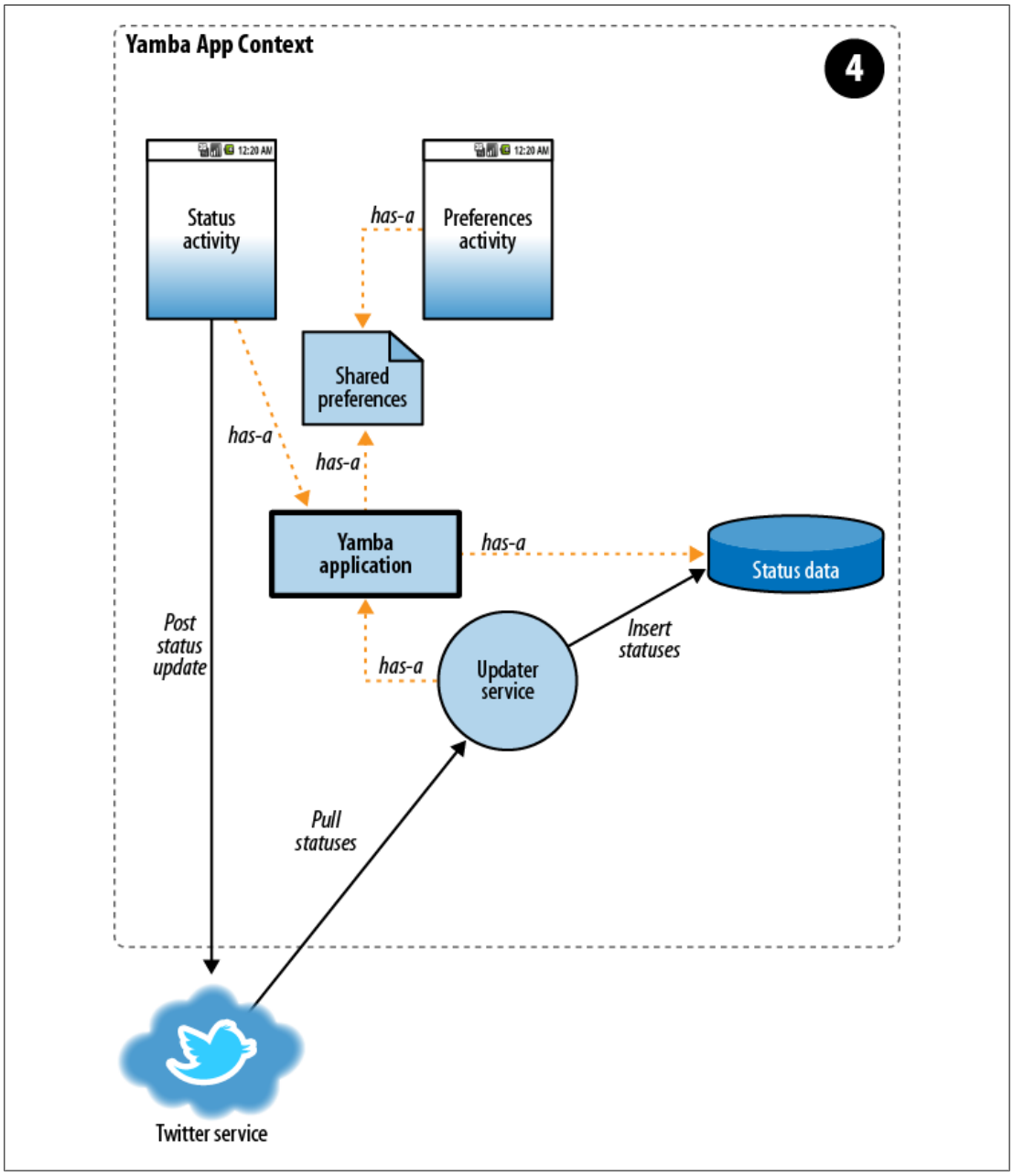


Figure 9-1. Yamba completion

CHAPTER 10
Lists and Adapters

In this chapter, you will learn how to create selection widgets, such as a `ListView`. But this isn't just a chapter about user interface elements. We are deepening our understanding of data from the previous chapter by learning how to read data from the status database and first simply output it to the screen as scrollable text. You will then learn about adapters in order to connect your database directly with the list and create a custom adapter to implement some additional functionality. You will link this new activity with your main activity so that the user can both post and read tweets.

By the end of this chapter, your app will be able to post new tweets, as well as pull them from Twitter, store them in the local database, and let the user read the statuses in a nice and efficient UI. At that point, your app will have three activities and a service.

TimelineActivity

We're going to create a new activity called `TimelineActivity` to display all the statuses from our friends. This activity pulls the data from the database and displays it on the screen. Initially, we do not have a lot of data in the database, but as we keep on using the application, the amount of statuses might explode. Our application needs to account for that.

We are going to build this activity in a few steps, keeping the application whole and complete as we make each improvement:

1. The first iteration of `TimelineActivity` uses a `TextView` to display all the output from the database. Since there may be quite a bit of data, we will use `ScrollView` to wrap our text and provide scroll bars.
2. The second iteration uses the much more scalable and efficient `ListView` and `Adapter` approach. In this step, you will learn how adapters and lists work.

3. Finally, we will create a custom `Adapter` to handle some additional business logic. At this point, we are going under the hood of an adapter and adding custom processing. You'll understand the purpose and usage of adapters better after this exercise.

Basic TimelineActivity Layout

In this first iteration, we are creating a new layout for the `TimelineActivity`. This layout initially uses a `TextView` to display all the data that we have in the database. This is fine initially when we don't have too many statuses to show.

Introducing ScrollView

Since it's unlikely that all our data will fit on a single page, we need a way to scroll the text. To do that, we use `ScrollView`. `ScrollView` is like a window that uses scroll bars to display part of a larger component that takes more space than the screen provides. To make some potentially large views scrollable, you wrap them with this `ScrollView`. For example, we have a printout of friends' statuses in the form of a `TextView`. As more statuses are added, this `TextView` could become large. In order to make it scrollable on a small screen, we put it into a `ScrollView`.

A `ScrollView` can contain only one direct child. If you want to combine multiple views into a single view that scrolls, you first need to organize those views into another layout, like you did previously in "The StatusActivity Layout" on page 52, and than add that layout into the `ScrollView`.

Typically you want `ScrollView` to take all the available space on the screen, so you will specify its layout width and height as `fill_parent`.

A `ScrollView` usually is not manipulated from Java, so it doesn't require an `id`.

In Example 10-1, we wrap our `TextView` with a `ScrollView` so that when there's a lot of text to display, `ScrollView` automatically adds scroll bars.

Example 10-1. res/layout/timeline_basic.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_height="fill_parent"
  android:layout_width="fill_parent" android:background="@drawable/background">

  <!-- Title ❶ -->
  <TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:layout_gravity="center"
    android:layout_margin="10dp" android:text="@string/titleTimeline"
    android:textColor="#fff" android:textSize="30sp" />

  <!-- Text output wrapper ❷ -->
  <ScrollView android:layout_height="fill_parent"
```

```
    android:layout_width="fill_parent">

    <!-- Text output ❸ -->
    <TextView android:layout_height="fill_parent"
      android:layout_width="fill_parent" android:id="@+id/textTimeline"
      android:background="#6000" />
  </ScrollView>

</LinearLayout>
```

❶ This is the title that we show at the top of this activity's screen. Notice that we defined the `titleTimeline` string resource in the */res/values/strings.xml* file, just like we did before in "Strings Resource" on page 55.

❷ The `ScrollView` that wraps our `TextView` and adds scroll bars as needed.

❸ The `TextView` that shows the actual text, in this case our friends' statuses from the database.

Creating the TimelineActivity Class

Now that we have the layout file, we need to create the `TimelineActivity` class. Just as with any other Java file, go to the Eclipse Package Explorer, right-click on your `com.marakana.yamba` package, choose New→Class, and name it `TimelineActivity`.

And just as before, whenever we create a new Java class that is also a main building block—an activity, service, broadcast receiver, or content provider—we first subclass a base class provided by the Android framework. In the case of activities, that class is `Activity`.

The method we almost universally override in any activity is `onCreate()`. This is a great place for us to initialize the database. The flip side of the coin is `onDestroy()`, a good place to clean up anything that we create in `onCreate()`. In this case, we close the database in `onDestroy()`. Because we'd like the data to be as fresh as possible, we put the code for querying the database and outputting the data in `onResume()`, the method called every time this activity is brought up front. Example 10-2 shows our code.

Example 10-2. TimelineActivity.java, version 1

```
package com.marakana.yamba5;

import android.app.Activity;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.widget.TextView;

public class TimelineActivity1 extends Activity { // ❶
  DbHelper dbHelper;
  SQLiteDatabase db;
  Cursor cursor;
  TextView textTimeline;
```

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

    // Find your views
    textTimeline = (TextView) findViewById(R.id.textTimeline);

    // Connect to database
    dbHelper = new DbHelper(this);  // ❷
    db = dbHelper.getReadableDatabase();  // ❸
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    // Close the database
    db.close(); // ❹
  }

  @Override
  protected void onResume() {
    super.onResume();

    // Get the data from the database
    cursor = db.query(DbHelper.TABLE, null, null, null, null, null,
        DbHelper.C_CREATED_AT + " DESC"); // ❺
    startManagingCursor(cursor);  // ❻

    // Iterate over all the data and print it out
    String user, text, output;
    while (cursor.moveToNext()) {  // ❼
      user = cursor.getString(cursor.getColumnIndex(DbHelper.C_USER));  // ❽
      text = cursor.getString(cursor.getColumnIndex(DbHelper.C_TEXT));
      output = String.format("%s: %s\n", user, text); // ❾
      textTimeline.append(output); // ❿
    }
  }


}
```

❶ This is an activity, so we start by subclassing the Android framework's `Activity` class.

❷ We need access to the database to get the timeline data. `onCreate()` is a good place to connect to the database.

❸ Once `dbHelper` opens the database file, we need to ask it for the actual database object. To do that, we can use either `getReadableDatabase()` or `getWritableDatabase()`. In this case, we are only reading the data from the timeline, so we open the database for reading only.

❹ At some point we need to close the database and release that resource. If the database was opened in `onCreate()`, the counterpart to that would be `onDestroy()`. So, we close the database there. Remember that `onDestroy()` is called only when the system has to free up resources.

❺ To query the data from the database, we use the `query()` method. This method seems to contain almost endless parameters, but most of them map nicely to various parts of the SQL `SELECT` statement. So this line is equivalent to SQL's `SELECT * FROM timeline ORDER BY created_at DESC`. The various `null` values refer to parts of the `SELECT` statement we are not using, such as `WHERE`, `GROUPING`, and `HAVING`. The data returned to us is of type `Cursor`, which is an iterator.

❻ `startManagingCursor()` is a convenience method that tells the activity to start managing the cursor's life cycle the same way it manages its own. This means that when this activity is about to be destroyed, it will make sure to release any data referred to by the cursor, thus helping Java's garbage collector clean up memory more quickly. The alternative is for us to add code manually in various override methods and worry about cursor management ourselves.

❼ `cursor`, if you recall from "Cursors" on page 122, represents all the data we received from the database `SELECT` statement that was effectively executed by our `query()` method. This data is generally in the form of a table, with many rows and columns. Each row represents a single record, such as a single status in our timeline. Each row also has columns that we predefined, such as `_id`, `created_at`, `user`, and `txt`. As we mentioned before, `cursor` is an iterator, meaning we can step through all its data one record at a time. The first call to `cursor`'s `moveToNext()` positions the cursor at the start. `moveToNext()` stops when there's no more data to process.

❽ For each record that the cursor currently points to, we can ask for its value by type and column index. So `cursor.getString(3)` returns a string value of the status, and `cursor.getLong(1)` gives us the timestamp indicating when this record was created. Refer back to Chapter 9 to see how we define strings such as `C_USER` and `C_TEXT` in our program that map to column names in the database. However, having hardcoded column indices is not a good practice, because if we ever change the schema, we'll have to remember to update this code. Also, the code is not very readable in this form. A better practice is to ask the database for the index of each column. We do that with the `cursor.getColumnIndex()` call.

❾ We use `String.format()` to format each line of the output. Because we chose the `TextView` widget to display the data, we can only display text, or in other words, formatted strings. In a later iteration of this code, we'll improve on this.

❿ We finally append that new line of output to our text view `textTimeline` so the user can see it on the screen.

Although this approach works for smaller data sets, it is not optimal or recommended. The better approach is to use a `ListView` to represent the list of statuses stored in the

database. `ListView`, which we'll use in the next version of our `TimelineActivity`, is much more scalable and efficient.

About Adapters

A `ScrollView` will work for a few dozen records. But what if your status database has hundreds or even thousands of records? Waiting to get and print them all would be highly inefficient. The user probably doesn't even care about all of the data anyhow.

To address this issue, Android provides adapters. These are a smart way to connect a `View` with some kind of data source (see Figure 10-1). Typically, your view would be a `ListView` and the data would come in the form of a `Cursor` or `Array`. So adapters come as subclasses of `CursorAdapter` or `ArrayAdapter`.

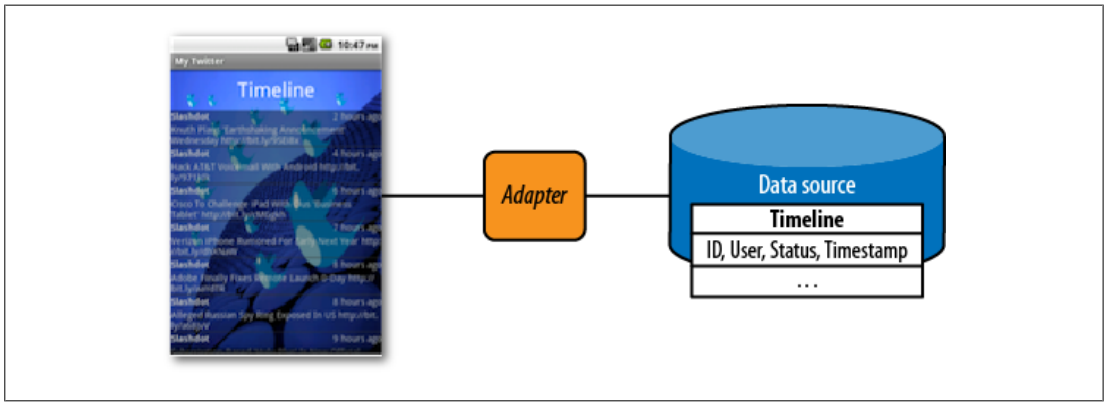


Figure 10-1. Adapter

Adding a ListView to TimelineActivity

As before, our first stop in upgrading our applications is our resources file. We'll add a `ListView` to the timeline layout by editing *timeline.xml*, shown in Example 10-3.

Example 10-3. res/layout/timeline.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_height="fill_parent"
  android:layout_width="fill_parent" android:background="@drawable/background">
  <TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:layout_gravity="center"
    android:layout_margin="10dp" android:text="@string/titleTimeline"
    android:textColor="#fff" android:textSize="30sp" />

  <!-- ❶ -->
  <ListView android:layout_height="fill_parent"
    android:layout_width="fill_parent" android:id="@+id/listTimeline"
    android:background="#6000" />
```

```
</LinearLayout>
```

❶ Adding `ListView` to your layout is like adding any other widget. The main attributes are `id`, `layout_height`, and `layout_width`.

ListView versus ListActivity

We could have used `ListActivity` as the parent class for our `TimelineActivity`. `ListActivity` is an activity that has a `ListView`. Either approach would work, but we chose to subclass `Activity` and create `ListView` separately to provide step-by-step, incremental learning.

`ListActivity` is slightly easier to use in cases where the built-in `ListView` is the only widget in the activity. `ListActivity` also makes it very easy to assign an existing array of elements to its list via the XML binding. However, we are using a `Cursor` for data and not an array (because our data comes from the database), and we do have an additional `TextView` for the scrollview's title, so the simplicity of `ListActivity` in this case is outweighed by the customization we require.

Creating a Row Layout

There's one more XML file to take care of. Although *timeline.xml* describes the entire activity, we also need to specify what a single row of data looks like—that is, a single line item on the screen that will show information such as who said what and when.

The easiest way to do that is to create another XML file just for that row. As for any new XML file, we use the Android New XML File dialog window: File→New→Android New XML File. Let's name this file *row.xml* and select `Layout` for the type.

For this layout, we chose one `LinearLayout` with two lines arranged vertically. The first line consists of the user and timestamp, and the second contains the actual status message. Notice that the first line uses another `LinearLayout` to position the user and timestamp horizontally next to each other.

The row of data in the `ListView` is represented by a custom layout defined in the *row.xml* file, shown in Example 10-4.

Example 10-4. res/layout/row.xml

```
<?xml version="1.0" encoding="utf-8"?>
  <!-- ❶ -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_height="wrap_content" android:orientation="vertical"
  android:layout_width="fill_parent">

  <!-- ❷ -->
  <LinearLayout android:layout_height="wrap_content"
    android:layout_width="fill_parent">
```

```
    <!-- ❸ -->
    <TextView android:layout_height="wrap_content"
      android:layout_width="fill_parent" android:layout_weight="1"
      android:id="@+id/textUser" android:text="Slashdot"
      android:textStyle="bold" />

    <!-- ❹ -->
    <TextView android:layout_height="wrap_content"
      android:layout_width="fill_parent" android:layout_weight="1"
      android:gravity="right" android:id="@+id/textCreatedAt"
      android:text="10 minutes ago" />
  </LinearLayout>

  <!-- ❺ -->
  <TextView android:layout_height="wrap_content"
    android:layout_width="fill_parent" android:id="@+id/textText"
    android:text="Firefox comes to Android" />

</LinearLayout>
```

❶ The main layout for the entire row. It is vertical because our row consists of two lines.

❷ A layout that runs horizontally and represents the first line of data, namely the user and timestamp.

❸ The user who posted this update.

❹ The timestamp indicating when it was posted. It should be a relative time (e.g., 10 minutes ago).

❺ The actual status.

Creating an Adapter in TimelineActivity.java

Now that we have the XML files sorted out, we are ready to update the Java code, shown in Example 10-5. First, we need to create the adapter. Adapters generally come in two flavors: those that represent array data and those that represent cursor data. Since our data is coming from the database, we are going to use the cursor-based adapter. One of the simplest of those is `SimpleCursorAdapter`.

`SimpleCursorAdapter` requires us to describe a single row of data (which we do in *row.xml*), the data (a cursor in our case), and the mapping for a single record of data to the single row in the list. The last parameter maps each cursor column to a view in the list.

Example 10-5. TimelineActivity.java, version 2

```
package com.marakana.yamba5;

import android.app.Activity;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
```

```
import android.widget.ListView;
import android.widget.SimpleCursorAdapter;

public class TimelineActivity2 extends Activity {
  DbHelper dbHelper;
  SQLiteDatabase db;
  Cursor cursor;  // ❶
  ListView listTimeline;  // ❷
  SimpleCursorAdapter adapter;  // ❸
  static final String[] FROM = { DbHelper.C_CREATED_AT, DbHelper.C_USER,
      DbHelper.C_TEXT };  // ❹
  static final int[] TO = { R.id.textCreatedAt, R.id.textUser, R.id.textText }; // ❺

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

    // Find your views
    listTimeline = (ListView) findViewById(R.id.listTimeline);  // ❻

    // Connect to database
    dbHelper = new DbHelper(this);
    db = dbHelper.getReadableDatabase();
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    // Close the database
    db.close();
  }

  @Override
  protected void onResume() {
    super.onResume();

    // Get the data from the database
    cursor = db.query(DbHelper.TABLE, null, null, null, null, null,
        DbHelper.C_CREATED_AT + " DESC");
    startManagingCursor(cursor);

    // Set up the adapter
    adapter = new SimpleCursorAdapter(this, R.layout.row, cursor, FROM, TO);  // ❼
    listTimeline.setAdapter(adapter); // ❽
  }

}
```

❶ Cursor to all the status updates that we have in the database.

❷ `listTimeline` is our `ListView` that displays the data.

❸ `adapter` is our custom adapter, explained in the text that follows this example.

❹ `FROM` is a string array specifying which columns in the cursor we're binding from. We use the same strings already used to refer to columns in our program.

❺ `TO` is an array of integers representing IDs of views in the *row.xml* layout to which we are binding data. The number of elements in `FROM` and `TO` must be the same, so that element at index 0 in `FROM` maps to element 0 in `TO`, and so on.

❻ We get the `ListView` from the XML layout.

❼ Once we have the data as a cursor, the layout of a single row from the *row.xml* file, and the `FROM` and `TO` constants for mapping the data, we are ready to create the `SimpleCursorAdapter`.

❽ Finally, we need to tell our `ListView` to use this adapter.

At this point, `TimelineActivity` is complete, but not yet registered with the manifest file. We'll do that in the next section. However, if we were to run this activity, you'd quickly notice that the timestamp doesn't look quite the way we imagined it.

Remember that we are storing the status creation time in the database as a `long` value representing the number of milliseconds since January 1st, 1970. And since that's the value in the database, that's the value we show on the screen as well. This is the standard Unix time, which is very useful for representing actual points in time. But the value is not very meaningful to users. Instead of showing value `1287603266359`, it would be much nicer to represent it to the user as "10 Minutes Ago." This friendly time format is known as *relative time*, and Android provides a method to convert from one format to the other.

The question is where to inject this conversion. As it stands right now, the `SimpleCursorAdapter` is capable only of mapping straight from a database value to layout view. This doesn't work for our needs, because we need to add some business logic in between the data and the view. To do this, we'll create our own adapter.

TimelineAdapter

`TimelineAdapter` is our custom adapter, shown in Example 10-6. Although `SimpleCursorAdapter` did a straightforward mapping of data in the database to views on the screen, we had an issue with the timestamp. The job of `TimelineAdapter` is to inject some business logic to convert the Unix timestamp to relative time. The method in `SimpleCursorAdapter` that creates a displayable view from input data is `bindView()`, so we'll override that method and ask it to massage the data before it is displayed.

Typically, if you are not sure which method to override, look at the online documentation for the particular system class that you are modifying (in this case, *http://developer.android.com/reference/android/widget/SimpleCursorAdapter.html*).

Example 10-6. TimelineAdapter.java

```
package com.marakana.yamba5;
```

```
import android.content.Context;
import android.database.Cursor;
import android.text.format.DateUtils;
import android.view.View;
import android.widget.SimpleCursorAdapter;
import android.widget.TextView;

public class TimelineAdapter extends SimpleCursorAdapter { // ❶
  static final String[] FROM = { DbHelper.C_CREATED_AT, DbHelper.C_USER,
      DbHelper.C_TEXT }; // ❷
  static final int[] TO = { R.id.textCreatedAt, R.id.textUser, R.id.textText }; // ❸

  // Constructor
  public TimelineAdapter(Context context, Cursor c) { // ❹
    super(context, R.layout.row, c, FROM, TO);
  }

  // This is where the actual binding of a cursor to view happens
  @Override
  public void bindView(View row, Context context, Cursor cursor) { // ❺
    super.bindView(row, context, cursor);

    // Manually bind created at timestamp to its view
    long timestamp = cursor.getLong(cursor
        .getColumnIndex(DbHelper.C_CREATED_AT)); // ❻
    TextView textCreatedAt = (TextView) row.findViewById(R.id.textCreatedAt); // ❼
    textCreatedAt.setText(DateUtils.getRelativeTimeSpanString(timestamp)); // ❽
  }

}
```

❶ To create our own custom adapter, we subclass one of the Android standard adapters, in this case the same `SimpleCursorAdapter` we used in the previous section.

❷ This constant defines the columns of interest to us in the database, as in the previous example.

❸ This constant specifies the IDs of views that we'll map those columns to.

❹ Because we're defining a new class, we need a constructor. It simply calls the parent constructor using `super`.

❺ The only method we override is `bindView()`. This method is called for each row to map its data to its views, and it's where the gist of the adapter work happens. In order to reuse most of the data-to-views mapping provided by `SimpleCursorAdapter`, we call `super.bindView()` first.

❻ To override default mapping for the timestamp, we first get the actual timestamp value from the database.

❼ Next, we find the specific `TextView` in the *row.xml* file.

❽ Finally, we set the value of `textCreatedAt` to the relative time since the timestamp. To do this, we use the Android SDK method `DateUtils.getRelativeTimeSpanString()`.

At this point, we can further simplify our `TimelineActivity` class because we moved some of the adapter details to `TimelineAdapter`. Example 10-7 shows this simplified code.

Example 10-7. TimelineActivity.java, version 3

```
package com.marakana.yamba5;

import android.app.Activity;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.widget.ListView;

public class TimelineActivity3 extends Activity {
  DbHelper dbHelper;
  SQLiteDatabase db;
  Cursor cursor;
  ListView listTimeline;
  TimelineAdapter adapter;  // ❶

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

    // Find your views
    listTimeline = (ListView) findViewById(R.id.listTimeline);

    // Connect to database
    dbHelper = new DbHelper(this);
    db = dbHelper.getReadableDatabase();
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    // Close the database
    db.close();
  }

  @Override
  protected void onResume() {
    super.onResume();

    // Get the data from the database
    cursor = db.query(DbHelper.TABLE, null, null, null, null, null,
        DbHelper.C_CREATED_AT + " DESC");
    startManagingCursor(cursor);
```

```
    // Create the adapter
    adapter = new TimelineAdapter(this, cursor);  // ❷
    listTimeline.setAdapter(adapter); // ❸
  }

}
```

❶ We change `SimpleCursorAdapter` to `TimelineAdapter`.

❷ Create a new instance of the `TimelineAdapter`, and pass it the context and the data.

❸ Set our `ListView` to connect to the data via the adapter.

One of the shortcomings of overriding `bindView()` is that we use `super.bindView()` to bind all views first, and then replace its behavior for one particular element. This is somewhat wasteful. The final version of our application in this chapter will optimize the process.

ViewBinder: A Better Alternative to TimelineAdapter

Instead of creating a new `TimelineAdapter` that is a subclass of `SimpleCursorAdapter` and overriding its `bindView()` method, we could attach the business logic directly to the existing `SimpleCursorAdapter`. This approach is more efficient because we are not overriding `bindView()` and we do not require a separate custom adapter class.

To attach business logic to an existing `SimpleCursorAdapter`, use its `setViewBinder()` method. We will need to supply the method with an implementation of `ViewBinder`. `ViewBinder` is an interface that specifies `setViewValue()`, where the actual binding of a particular date element to a particular view happens.

Again, we discovered the `setViewBinder()` feature of this `SimpleCursorAdapter` framework class by reading its reference documentation.

In our final iteration of `TimelineAdapter`, we create a custom `ViewBinder` as a constant and attach it to the stock `SimpleCursorAdapter`, as shown in Example 10-8.

Example 10-8. TimelineActivity.java with ViewBinder

```
  ...

  @Override
  protected void onResume() {
    ...
    adapter.setViewBinder(VIEW_BINDER); // ❶
    ...
  }

  // View binder constant to inject business logic that converts a timestamp to
  // relative time
  static final ViewBinder VIEW_BINDER = new ViewBinder() { // ❷
```

```
    public boolean setViewValue(View view, Cursor cursor, int columnIndex) { // ❸
      if (view.getId() != R.id.textCreatedAt)
        return false; // ❹

      // Update the created at text to relative time
      long timestamp = cursor.getLong(columnIndex); // ❺
      CharSequence relTime = DateUtils.getRelativeTimeSpanString(view
          .getContext(), timestamp); // ❻
      ((TextView) view).setText(relTime); // ❼

      return true; // ❽
    }

  };

  ...
```

❶ We attach a custom `ViewBinder` instance to our stock adapter. `VIEW_BINDER` is defined later in our code.

❷ The actual implementation of a `ViewBinder` instance. Notice that we are implementing it as an inner class. There's no reason for any other class to use it, and thus it shouldn't be exposed to the outside world. Also notice that it is `static final`, meaning that it's a constant.

❸ The only method that we need to provide is `setViewValue()`. This method is called for each data element that needs to be bound to a particular view.

❹ First we check whether this view is the view we care about, i.e., our `TextView` representing when the status was created. If not, we return `false`, which causes the adapter to handle the bind itself in the standard manner. If it is our view, we move on and do the custom bind.

❺ We get the raw timestamp value from the cursor data.

❻ Using the same Android helper method we used in our previous example, `DateUtils.getRelativeTimeSpanString()`, we convert the timestamp to a human-readable format. This is that business logic that we are injecting.

❼ Update the text on the actual view.

❽ Return `true` so that `SimpleCursorAdapter` does not process `bindView()` on this element in its standard way.

Updating the Manifest File

Now that we have the `TimelineActivity`, it would make sense to make it the "main" activity for our Yamba application. After all, users are more likely to check what their friends are doing than to update their own status.

To do that, we need to update the manifest file. As usual, we'll list `TimelineActivity` within the `<activity>` element in the *AndroidManifest.xml* file, just as we added the preference activity to the manifest file in "Update the Manifest File" on page 88:

```
<activity android:name=".TimelineActivity" />
```

Now, in order to make `TimelineActivity` the main entry point into our application, we need to register it to respond to certain *intents*. Basically, when the user clicks to start your application, the system sends an intent. You have to define an activity to "listen" to this intent. The activity does that by filtering the intents with an `IntentFilter`. In XML, this is within the `<intent-filter>` element, and it usually contains at least an `<action>` element representing the actual intent action we're interested in.

You might have noticed that `StatusActivity` had some extra XML compared to `PrefsActivity`. The extra code is the intent filter block, along with the action that it's filtering for.

There is a special action named `android.intent.action.MAIN` that simply indicates this is the main component that should be started when the user wants to start your application. Additionally, the `<category>` element tells the system that this application should be added to the main Launcher application so that the user can see its app icon along with all the other icons, click on it, and start it. This category is defined as `android.intent.category.LAUNCHER`.

So, to make `TimelineActivity` the main entry point, we simply list it and move the code from the `StatusActivity` declaration over to the `TimelineActivity` declaration, as shown in Example 10-9.

Example 10-9. AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1" android:versionName="1.0" package="com.marakana.yamba5">
  <application android:icon="@drawable/icon" android:label="@string/app_name"
    android:name=".YambaApplication">

    <activity android:name=".TimelineActivity" android:label="@string/titleTimeline">
      <intent-filter> <!-- ❶ -->
        <action android:name="android.intent.action.MAIN" /> <!-- ❷ -->
        <category android:name="android.intent.category.LAUNCHER" /> <!-- ❸ -->
      </intent-filter>
    </activity>

    <activity android:name=".PrefsActivity" android:label="@string/titlePrefs" />
    <activity android:name=".StatusActivity"
              android:label="@string/titleStatus" /> <!-- ❹ -->

    <service android:name=".UpdaterService" />

  </application>
  <uses-sdk android:minSdkVersion="8" />
```

```
  <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

❶ `<intent_filter>` registers this particular activity with the system to respond to certain intents.

❷ Tells the system that this is the main activity to start when users start your application.

❸ The category `LAUNCHER` tells the Home application to add this application into the list displayed in the launcher drawer.

❹ `StatusActivity` no longer needs any intent filters.

Initial App Setup

Now when the user runs our application, the Timeline screen will show up first. But unless the user knows she should set up the preferences and start the service, there will be no data and very little hand-holding telling her what to do.

One solution is to check whether preferences exist, and if they do not, redirect the user to the Preference activity with a message telling her what to do next:

```
...
@Override
protected void onCreate(Bundle savedInstanceState) {
  ...
  // Check whether preferences have been set
  if (yamba.getPrefs().getString("username", null) == null) { // ❶
    startActivity(new Intent(this, PrefsActivity.class)); // ❷
    Toast.makeText(this, R.string.msgSetupPrefs, Toast.LENGTH_LONG).show(); // ❸
  }
  ...
}
...
```

❶ We check whether a particular preference has been set. In this case, I've chosen to check `username` because it's likely to be set if any preferences at all are set. Since the preferences do not exist the first time the user runs the application, this means the value of `username` (or any other preference item we choose) will be `null`.

❷ We start the `PrefsActivity`. Note that `startActivity()` will dispatch an intent to the system, but the rest of `onCreate()` will execute as well. This is good because we're likely going to come back to the Timeline activity once we're done setting up preferences.

❸ We display a little pop-up message, i.e., a `Toast`, telling the user what to do. This assumes that you have created the appropriate `msgSetupPrefs` in your *strings.xml* file.

Base Activity

Now that we have a Timeline activity, we need to give it an options menu, just as we did for our Status activity in "The Options Menu" on page 89. This is especially important because the Timeline activity is the entry point into our application, and without the menu, the user cannot easily get to any other activity or start and stop the service.

As one approach, we could copy and paste the code we already have from the Status activity, but that's rarely a good strategy. Instead, we'll do what we usually do: refactor the code. In this case, we can take out the common functionality from the Status activity and place it in another activity that will serve as the base. See Figure 10-2.

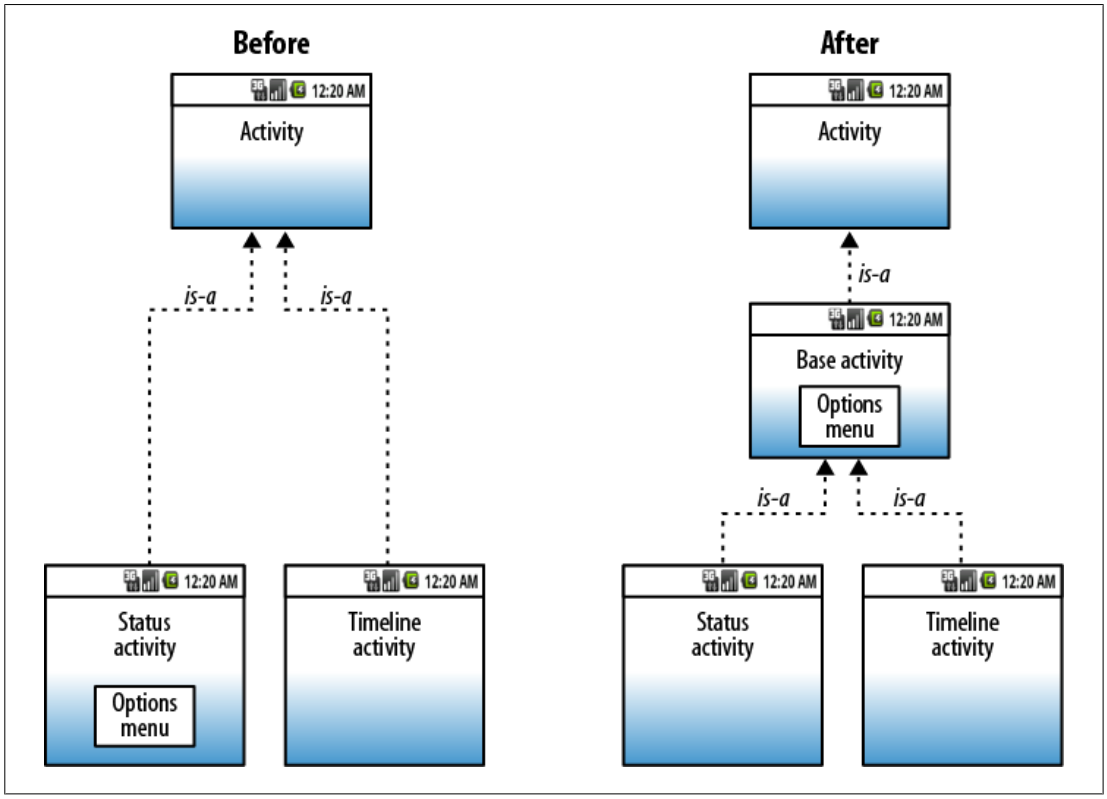


Figure 10-2. BaseActivity refactor

To do that, we'll create a new class called `BaseActivity` and move the common functionality into it. For us, the common functionality includes getting the reference to the `YambaApplication` object, as well as the `onCreateOptionsMenu()` and `onOptionsItemSelected()` methods that support the options menu.

Toggle Service

While we're at it, instead of having Start Service and Stop Service menu buttons, it would be nice to provide just one button that toggles between Start and Stop. To do that, we'll change our menu and add `onMenuOpened()` to the base activity to dynamically update the title and images for this toggle item.

First, we'll update the *menu.xml* file to include our new toggle menu item, as shown in Example 10-10. At the same time, we'll remove the Start Service and Stop Service items because our toggle feature makes them obsolete.

Example 10-10. res/menu/menu.xml[]

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item android:id="@+id/itemStatus" android:title="@string/titleStatus"
    android:icon="@android:drawable/ic_menu_edit"></item>
  <item android:title="@string/titleTimeline" android:id="@+id/itemTimeline"
    android:icon="@android:drawable/ic_menu_sort_by_size"></item>
  <item android:id="@+id/itemPrefs" android:title="@string/titlePrefs"
    android:icon="@android:drawable/ic_menu_preferences"></item>
  <item android:icon="@android:drawable/ic_menu_delete"
    android:title="@string/titlePurge" android:id="@+id/itemPurge"></item>

  <!-- ❶ -->
  <item android:id="@+id/itemToggleService" android:title="@string/titleServiceStart"
    android:icon="@android:drawable/ic_media_play"></item>

</menu>
```

❶ This new `itemToggleService` now replaces both `itemServiceStart` and `itemServiceStop`.

Next, we need to override `onMenuOpened()` in the base activity to change the menu item dynamically, shown in Example 10-11.

Example 10-11. BaseActivity.java

```
package com.marakana.yamba5;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.Toast;

/**
 * The base activity with common features shared by TimelineActivity and
 * StatusActivity
 */
public class BaseActivity extends Activity { // ❶
  YambaApplication yamba; // ❷
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  yamba = (YambaApplication) getApplication(); // ❸
}

// Called only once first time menu is clicked on
@Override
public boolean onCreateOptionsMenu(Menu menu) { // ❹
  getMenuInflater().inflate(R.menu.menu, menu);
  return true;
}

// Called every time user clicks on a menu item
@Override
public boolean onOptionsItemSelected(MenuItem item) { // ❺

  switch (item.getItemId()) {
  case R.id.itemPrefs:
    startActivity(new Intent(this, PrefsActivity.class)
        .addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT));
    break;
  case R.id.itemToggleService:
    if (yamba.isServiceRunning()) {
      stopService(new Intent(this, UpdaterService.class));
    } else {
      startService(new Intent(this, UpdaterService.class));
    }
    break;
  case R.id.itemPurge:
    ((YambaApplication) getApplication()).getStatusData().delete();
    Toast.makeText(this, R.string.msgAllDataPurged, Toast.LENGTH_LONG).show();
    break;
  case R.id.itemTimeline:
    startActivity(new Intent(this, TimelineActivity.class).addFlags(
        Intent.FLAG_ACTIVITY_SINGLE_TOP).addFlags(
        Intent.FLAG_ACTIVITY_REORDER_TO_FRONT));
    break;
  case R.id.itemStatus:
    startActivity(new Intent(this, StatusActivity.class)
        .addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT));
    break;
  }
  return true;
}

// Called every time menu is opened
@Override
public boolean onMenuOpened(int featureId, Menu menu) { // ❻
  MenuItem toggleItem = menu.findItem(R.id.itemToggleService); // ❼
  if (yamba.isServiceRunning()) { // ❽
    toggleItem.setTitle(R.string.titleServiceStop);
    toggleItem.setIcon(android.R.drawable.ic_media_pause);
  } else { // ❾
```

```
      toggleItem.setTitle(R.string.titleServiceStart);
      toggleItem.setIcon(android.R.drawable.ic_media_play);
    }
    return true;
  }

}
```

❶ `BaseActivity` is an `Activity`.

❷ We declare the shared `YambaApplication` to make it accessible to all the other subclasses.

❸ In `onCreate()`, we get the reference to `yamba`.

❹ `onCreateOptionsMenu()` is moved here from `StatusActivity`.

❺ `onOptionsItemSelected()` is also moved over from `StatusActivity`. Notice, however, that it now checks for `itemToggleService` instead of start and stop service items. Based on the state of the service, which we know from the flag in `yamba`, we request either to start or to stop the updater service.

❻ `onMenuOpened()` is the new method called by the system when the options menu is opened. This is a good callback for us to implement the toggle functionality. We're given the `menu` object that represents the options menu.

❼ Within the menu object, we find our new toggle item so that we can update it based on the current state of the updater service.

❽ We check whether the service is already running, and if it is, we set the appropriate title and icon for the toggle item. Notice that here we're setting up the title and icon programmatically using the Java APIs instead of the XML, which we used initially to set up the menu in *menu.xml*.

❾ If the service is stopped, we set the icon and title so that user can click on it and start the service. This way our single toggle button communicates the service's current state.

Now that we have a `BaseActivity` class, let's update our Timeline activity to use it. Example 10-12 shows what the completed Timeline activity looks like.

Example 10-12. TimelineActivity.java, final version

```
package com.marakana.yamba5;

import android.content.Intent;
import android.database.Cursor;
import android.os.Bundle;
import android.text.format.DateUtils;
import android.view.View;
import android.widget.ListView;
import android.widget.SimpleCursorAdapter;
import android.widget.TextView;
import android.widget.Toast;
```

```
import android.widget.SimpleCursorAdapter.ViewBinder;

public class TimelineActivity extends BaseActivity { // ❶
  Cursor cursor;
  ListView listTimeline;
  SimpleCursorAdapter adapter;
  static final String[] FROM = { DbHelper.C_CREATED_AT, DbHelper.C_USER,
      DbHelper.C_TEXT };
  static final int[] TO = { R.id.textCreatedAt, R.id.textUser, R.id.textText };

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

    // Check if preferences have been set
    if (yamba.getPrefs().getString("username", null) == null) { // ❷
      startActivity(new Intent(this, PrefsActivity.class));
      Toast.makeText(this, R.string.msgSetupPrefs, Toast.LENGTH_LONG).show();
    }

    // Find your views
    listTimeline = (ListView) findViewById(R.id.listTimeline);
  }

  @Override
  protected void onResume() {
    super.onResume();

    // Setup List
    this.setupList(); // ❸
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    // Close the database
    yamba.getStatusData().close(); // ❹
  }

  // Responsible for fetching data and setting up the list and the adapter
  private void setupList() { // ❺
    // Get the data
    cursor = yamba.getStatusData().getStatusUpdates();
    startManagingCursor(cursor);

    // Setup Adapter
    adapter = new SimpleCursorAdapter(this, R.layout.row, cursor, FROM, TO);
    adapter.setViewBinder(VIEW_BINDER); // ❻
    listTimeline.setAdapter(adapter);
  }

  // View binder constant to inject business logic for timestamp to relative
  // time conversion
```

```
  static final ViewBinder VIEW_BINDER = new ViewBinder() { // ❼

    public boolean setViewValue(View view, Cursor cursor, int columnIndex) {
      if (view.getId() != R.id.textCreatedAt)
        return false;

      // Update the created at text to relative time
      long timestamp = cursor.getLong(columnIndex);
      CharSequence relTime = DateUtils.getRelativeTimeSpanString(view
          .getContext(), timestamp);
      ((TextView) view).setText(relTime);

      return true;
    }

  };
}
```

❶ For starters, we now subclass our `BaseActivity` instead of just the system's `Activity`. This way we inherit the `yamba` object as well as all the support for the options menu.

❷ This is where we check whether preferences are already set. If not, we'll redirect the user to the Preference activity first.

❸ On resuming this activity, we set up the list. This is a private method, shown later in the code.

❹ When this activity is closed, we want to make sure we close the database to release this resource. The database is opened by the call to `getStatusUpdates()` in the `yamba` application.

❺ `setupList()` is the convenience method that gets the data, sets up the adapter, and connects it all to the list view.

❻ This is where we attach the view binder to the list, as discussed earlier in "ViewBinder: A Better Alternative to TimelineAdapter" on page 149.

❼ `ViewBinder` is defined here.

At this point, we've done a lot of the refactoring work on our Timeline activity. We can also simplify the Status activity by cutting out the code related to the options menu. This also helps separate functional concerns among `BaseActivity`, `StatusDate`, and `TimelineActivity`.

Figure 10-3 shows what the final Timeline activity screen looks like.

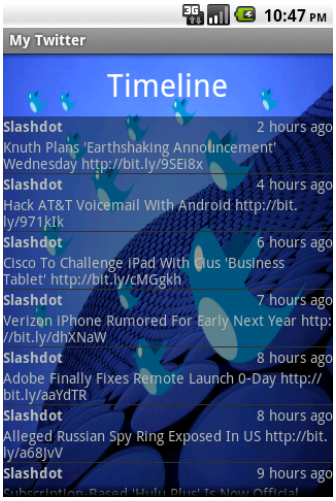


Figure 10-3. TimelineActivity

Summary

At this point, Yamba can post a new status as well as list the statuses of our friends. Our application is complete and usable.

Figure 10-4 illustrates what we have done so far as part of the design outlined earlier in Figure 5-4.

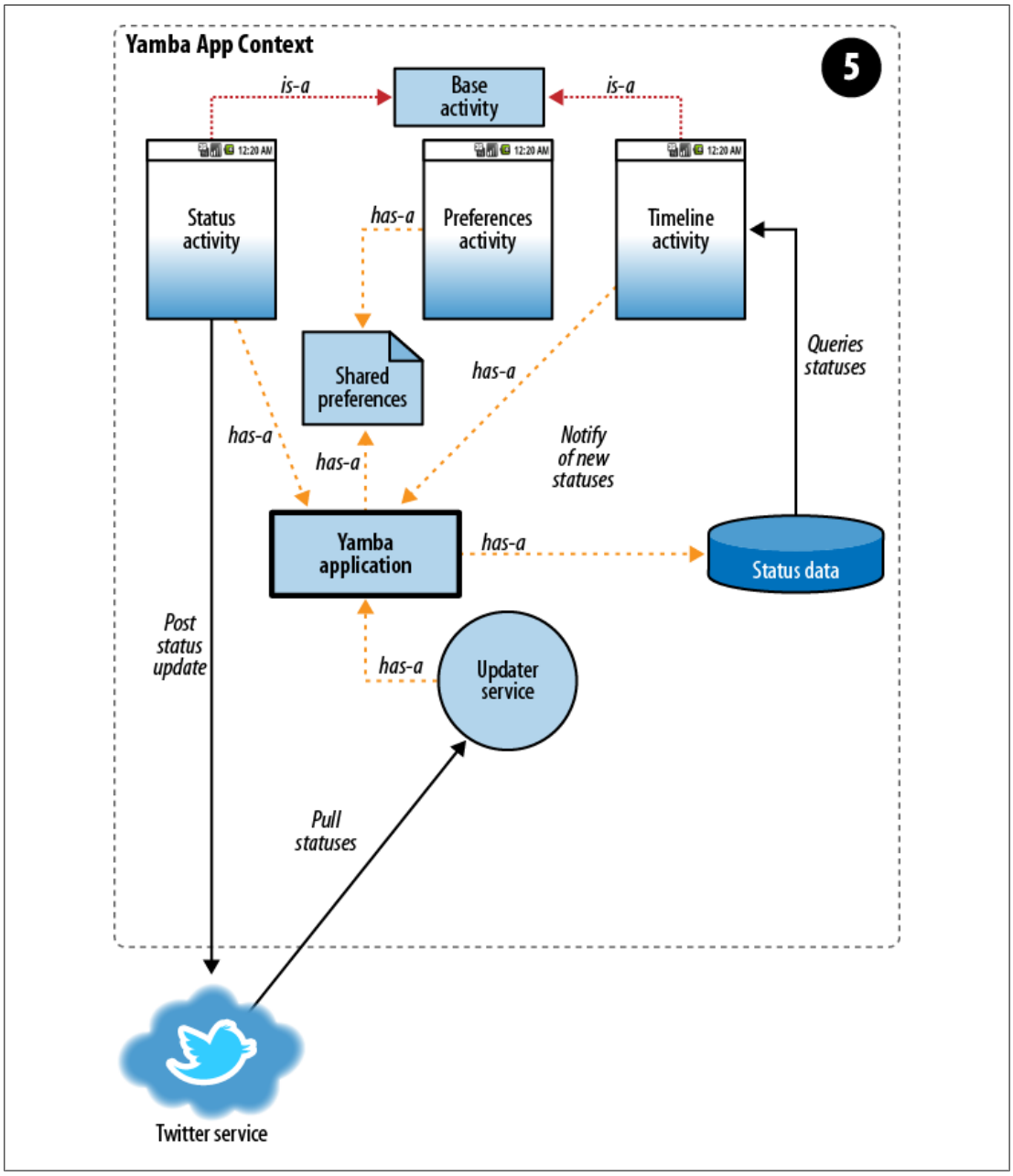


Figure 10-4. Yamba completion