III. Application Development:

Android app with Couchbase Lite

The app we are going to create is an events app that will have the information of an event where a user is able to update the fields accordingly.

1. We assume you are already familiar with the android app Activity life-cycle methods:

onCreate()

onStart()

onDestroy()

onPause()

onResume()

onRestart()

onStop()

If not, be sure to research on each of their functionality over at: Xxxxx

1. Before we modify the provided CouchbaseEvents code, we will explore the base level code to gain a better understanding. This application is based on events the user would create, such as ‘party’, or a ‘rock concert’ they wish to attended:

This code will extend an android.app.Activity:

package com.couchbase.training.couchbaseevents;

import java.io.ByteArrayInputStream;

import java.util.HashMap;

import java.util.Map;

import com.couchbase.lite.Attachment;

import com.couchbase.lite.CouchbaseLiteException;

import com.couchbase.lite.Database;

import com.couchbase.lite.Document;

import com.couchbase.lite.Manager;

import com.couchbase.lite.SavedRevision;

import com.couchbase.lite.UnsavedRevision;

import com.couchbase.lite.android.AndroidContext;

import com.couchbase.lite.util.Log;

import com.couchbase.training.couchbaseevents.R;

import android.support.v7.app.ActionBarActivity;

import android.os.Bundle;

import android.view.Menu;

import android.view.MenuItem;

public class HelloWorldActivity extends ActionBarActivity {

public static final String DB\_NAME = "couchbaseevents";

public static final String TAG = "couchbaseevents";

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

helloCBL();

}

C.

The onCreate() method is overridden to invoke our HelloCBL method, which is the focus of this tutorial. (action is) and where the primary funcationalities of Couchbase Lite resides.

private void helloCBL() {

Manager manager = null;

Database database = null;

try {

manager = new Manager(

new AndroidContext(this),

Manager.DEFAULT\_OPTIONS);

database = manager.getDatabase(DB\_NAME);

} catch (Exception e) {

Log.e(TAG, "Error getting database", e);

return;

}

// Create the document

String documentId = createDocument(database);

/\* Get and output the contents \*/

outputContents(database, documentId);

/\* Update the document and add an attachment \*/

updateDoc(database, documentId);

// Add an attachment

addAttachment(database, documentId);

/\* Get and output the contents with the attachment \*/

outputContentsWithAttachment(database, documentId);

}

D.

Including com.couchbase.lite.Manager class is where we begin. The AndroidContext and ManagerOptions is required to be passed to the constructor.

XXX: ADD SOMETHING VISUAL

E.

Instantiate the Manager class to use the reference to invoke the factory method ‘manager.getDatabase()’. From there you would obtain a com.couchbase.lite.Database which is provided through a database variable named, “couchbaseevennts” for the events planning and management app.

XXX: Add SOMETHING VISUAL

F. We will follow best practices and make the Manager and Database references available to the rest of the Android app as a Singleton. You may read more on why having a Singleton is a better approach over on the Couchbase Mobile Blog.

**public** Database getDatabaseInstance() **throws** CouchbaseLiteException{

**if** ((**this**.database == **null**) & (**this**.manager != **null**)) {

**if** (Manager.*isValidDatabaseName*(DB\_NAME) {

**this**.database =

manager.getDatabase(DB\_NAME);

}

}

**return** database;

}

**public** Manager getManagerInstance() **throws** IOException {

**if** (manager == **null**) {

manager = **new** Manager(**new** AndroidContext(**this**),

Manager.*DEFAULT\_OPTIONS*);

}

**return** manager;

}

G. The createDocument() method below is creating an … [explain what the code is doing

/\*\*

\* Creates the document

\*

\* @param database

\* The CBL database

\* @return

\* The Id of the Document that was created

\*/

private String createDocument(Database database) {

// Create a new document and add data

Document document = database.createDocument();

String documentId = document.getId();

Map<String, Object> map =

new HashMap<String, Object>();

map.put("name", "Big Party");

map.put("location", "My House");

try {

// Save the properties to the document

document.putProperties(map);

} catch (CouchbaseLiteException e) {

Log.e(TAG, "Error putting", e);

}

return documentId;

}

Explanation:

Go through the document ID to retrieve a document

Hashmap resembles a dictionary key value format and use the map to store properties that will be written to the Document. The name and location of the event. The putProperties method writes the contents of the map into the Document which actually saves the passed in parameter data from the Map to Couchbase Lite database on the local device

We return the documentID so that the caller has the document ID for their records

IV. **Updating an Existing Document, and adding Attachment(s).**

First we will explore the process of updating an existing document, and then we will look carefully at creating an Attachment to that document.

Attachments will currently only be made available to the Couchbase Lite API using them.

A. To update an existing document, you will need the key, let’s take a look:

/\*\*

\* Updates the document

\*

\* @param database

\* The CBL database

\* @param documentId

\* The Id of the Document to output

\*/

private void updateDoc(Database database,

String documentId) {

Document document = database.getDocument(documentId);

try {

// Update the document with more data

Map<String, Object> updatedProperties =

new HashMap<String, Object>();

updatedProperties.putAll(document.getProperties());

updatedProperties.put("eventDescription",

"Everyone is invited!");

updatedProperties.put("address", "123 Elm St.");

/\* Save to the Couchbase local Couchbase Lite DB \*/

document.putProperties(updatedProperties);

} catch (CouchbaseLiteException e) {

Log.e(TAG, "Error putting", e);

}

}

1. As you can see, we first need to retrieve the original document before we can update its content, and given the Database and documentId, we can retrieve the document.
2. Then we create a fresh Map to hold the properties (data) we wish to use to update the document.
   1. First we populate the Map with the existing data we have already in the document, this makes sense, as that data may be changed.
   2. Next we put new data into the properties Map, which we will save to the document shortly, such as the party description and address.
   3. Lastly we put the Map of properties back into the document as a set of properties and values with document.putProperties(…).this putProperties(…) method saves the data to the local Couchbase Lite database on the device
   4. IMPORTANT: It would be easy to think that all that happens here is that the Document object is updated with data, but in fact this causes that document to be stored in the local database. This is also different from the data being replicated to the Sync Gateway. So far, there is no Sync Gateway involved. We will cover that later. This is currently all local data on the device only.

**We can also add a binary attachment. As of now, this attachment can only be read by the Couchbase Lite (CBL) API.**

/\*\*

\* Adds an attachment

\*

\* @param database

\* The CBL database

\* @param documentId

\* The Id (key) of the Document

\*/

private void addAttachment(Database database,

String documentId) {

Document document = database.getDocument(documentId);

try {

/\* Add an attachment with sample data as POC \*/

ByteArrayInputStream inputStream = new

ByteArrayInputStream(

new byte[] { 0, 0, 0, 0 });

UnsavedRevision revision = document.getCurrentRevision()

.createRevision();

revision.setAttachment("binaryData",

"application/octet-stream", //MIME type

inputStream);

/\* Save doc & attachment to the local DB \*/

revision.save();

} catch (CouchbaseLiteException e) {

Log.e(TAG, "Error putting", e);

}

}

Explained

1. We start by fetching the document to which we wish to attach.
2. Then we create a ByteArrayInputStream to hold our binary (attachment) data. Of course this is mock data, but it satisfies our proof of concept.
3. Then we get the current revision of the document, since the document could have been modified since we last read it, and we want the latest and greatest data. We use the document factory methods to request that UnsavedRevision which we wish to update with an attachment.
4. The UnsavedRevision method setAttachment(…) let’s us place the stream we created on the Document. Note we need to pass three parameters:
   1. The name of the attachment (multiple attachments are allowed)
   2. The MIME (Multipurpose Internet Mail Extensions) type (see: http://en.wikipedia.org/wiki/Internet\_media\_type#List\_of\_common\_media\_types)
   3. The java.io.InputStream (or sub-type). The attachment data will be written to the Database when the UnsavedRevision is saved.
5. Then we invoke the save() method to save the Document and it’s attachment to the Database locally.
6. **Once we have a document with an attachment, we will want to retrieve the attachment using the CBL API. Let’s see how:**

Document fetchedSameDoc = getDatabaseInstance().getExistingDocument(getDocId());

SavedRevision saved = fetchedSameDoc.getCurrentRevision();

//The content of the attachment is a byte[] we created

Attachment attach = saved.getAttachment("binaryData");

**int** i = 0;

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(attach.getContent()));

StringBuffer values = **new** StringBuffer();

**while** (i++ < 4) { //We knew the size of the byte array

//This is the content of the attachment

values.append(reader.read() + " ");

}

Log.*v*("LaurentActivity", "The docID: " + getDocId() + ", attachment contents was: " + values.toString());

1. To better understand our code, let’s start with the Document fetchedSameDoc, which represents the document you want to grab with the attachment you just put there.
2. It is the same document, so given the document ID we can easily fetch it with the getExistingDocument(…) method of the Database instance.
3. From the document, we want to make sure we have the most current revision, and since we saved it previously, it is a SavedRevision instance.
4. As long as we know the unique name of the attachment on that SavedRevision, we can ask for the attachment using the getAttachment(“name”) method, to retrieve the Attachment object.
5. The Attachment getContent() method returns the InputStream we placed on the document as the actual attachment.
6. We can wrap the returned InputSteam with an InputSteamReader, and wrap a BufferedReader around that to read the data from the attachment into a StringBuffer for display.
7. In the real world, all this insert, update, read and delete work would be performed in some type of CrudRepository, such as EventRepository when working with Events.
8. Here is the Event class:

**package** com.example.couchbaseevents;

**public** **class** Event {

**public** Event(String name, String address, String description, String date, String time, String eventType) {

**super**();

**this**.name = name;

**this**.address = address;

**this**.description = description;

**this**.date = date;

**this**.time = time;

**this**.eventType = eventType;

System.*out*.println("\nWe have created a POJO Event...");

}

**public** Event() {

/\* For SerDe to and from JSON... \*/

}

**private** String name;

**private** String address;

**private** String description;

**private** String date;

**private** String time;

**private** String eventType;

**private** String \_id; // Document ID

**private** String \_rev; // Current revision

**private** String url; //Optional

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

**public** String getDescription() {

**return** description;

}

**public** **void** setDescription(String description) {

**this**.description = description;

}

**public** String getDate() {

**return** date;

}

**public** **void** setDate(String date) {

**this**.date = date;

}

**public** String getTime() {

**return** time;

}

**public** **void** setTime(String time) {

**this**.time = time;

}

**public** String getEventType() {

**return** eventType;

}

**public** **void** setEventType(String eventType) {

**this**.eventType = eventType;

}

**public** String get\_id() {

**return** \_id;

}

**public** **void** set\_id(String \_id) {

**this**.\_id = \_id;

}

**public** String get\_rev() {

**return** \_rev;

}

**public** **void** set\_rev(String \_rev) {

**this**.\_rev = \_rev;

}

**public** String getUrl() {

**return** url;

}

**public** **void** setUrl(String url) {

**this**.url = url;

}

}

1. If we have an Event, we would create an EventRepository with CRUD methods for each Document like:
2. public Event save(Event event) { /\* Create code \*/}
3. public Event getById(String docId) { /\* Read code \*/}
4. public Event update(Event event) { /\* Update code \*/}
5. public boolean delete(Event event) { /\* Delete code \*/}
6. public boolean exists(String docId) { /\* Exists code \*/}
7. Here is some example EventRepository update(…) code:

public Event update(Event event)

throws CouchbaseLiteException {

String docId = event.get\_id();

Document updateDoc = db.getExistingDocument(docId);

if (updateDoc != null) {

Map<String, Object> updateMap =

new HashMap<String, Object>();

/\* add Event data to updateMap \*/

updateMap.put("name", event.getName());

updateMap.put("address", event.getAddress());

updateMap.put("description",

event.getDescription());

updateMap.put("date", event.getDate());

updateMap.put("time", event.getTime());

updateMap.put("eventType", event.getEventType());

updateMap.put("\_id", docId);

updateMap.put("\_rev", event.get\_rev());

updateMap.put("url", event.getUrl());

/\* Saves this data to local Couchbase Lite db \*/

SavedRevision revision =

updateDoc.putProperties(updateMap);

/\* here we set the revision id value... \*/

event.set\_rev(updateDoc.getCurrentRevisionId());

}

return event;

}