Couchbase Lite (Mobile) Tutorial for Android

by Laurent Weichberger

(Version 7 ~ Revised: Feb. 20, 2015)

This Couchbase Lite Tutorial is based on Android code written in Eclipse Kepler (IDE) on a MacBook Pro (16 GB RAM), with Couchbase Lite version 1.0.3.1. Throughout the tutorial we will reference screen shots and Java code. If you are using another OS, and IDE or language, we will share more about iOS/Xcode projects in a subsequent tutorial.

I. Setup your Environment for Success.

The first thing we need to do is get all of our software in place, and configured properly. We will be using the Couchbase Lite 1.0.3 download, as well as Eclipse and Android. Some of the jar files will need to be moved to their proper places after download, so we have given you step by step instructions on how to install all the software properly before we start developing.

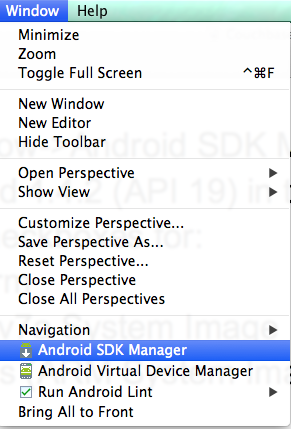
A. Let’s start with Eclipse.

1. Make sure Eclipse 4.4.x or more recent is being used.

2. Within Eclipse use the menu option: Help > Install New Software to install the “ADT Plugin” for Android development: https://dl-ssl.google.com/android/eclipse/

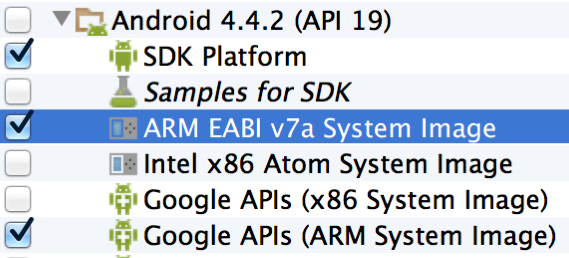
3. If you have difficulty with this step, go here: http://developer.android.com/sdk/installing/installing­adt.html

4. If you do this properly, you will have a new selection from the menu in Eclipse > Window > Android SDK Manager:



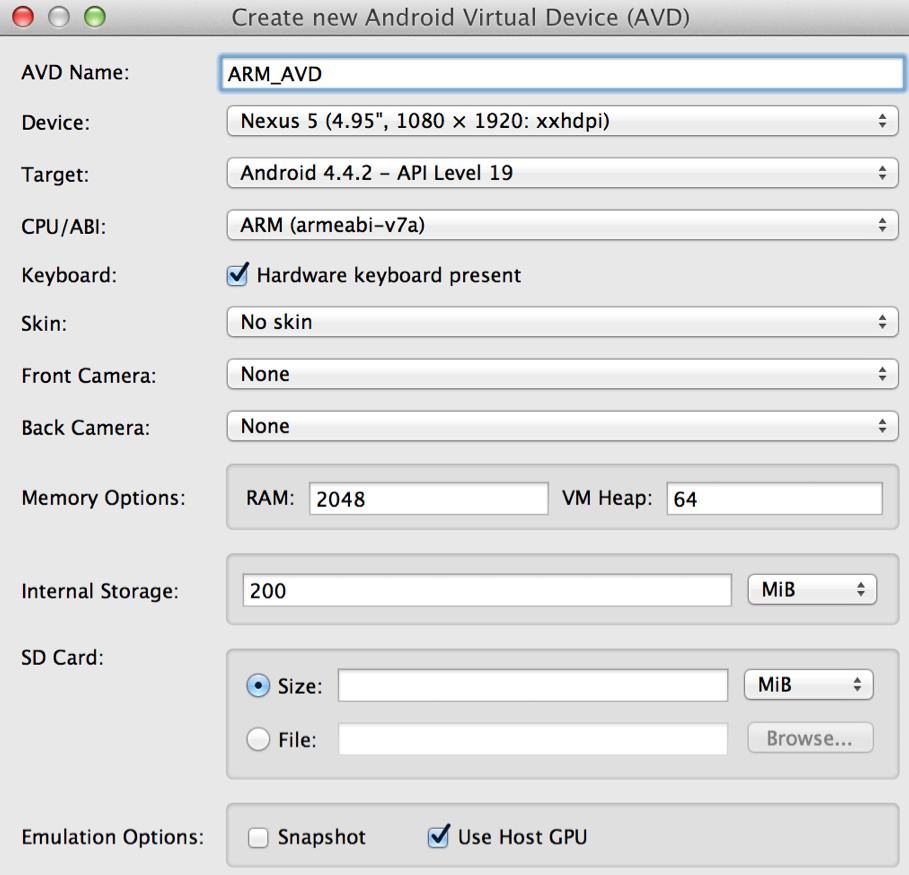
5. Find Android 4.4.2 (API 19) in the list and click the checkboxes for:

SDK Platform & ARM EABI v7a System Image & Google APIs (ARM System Image), and click “Install Packages”:

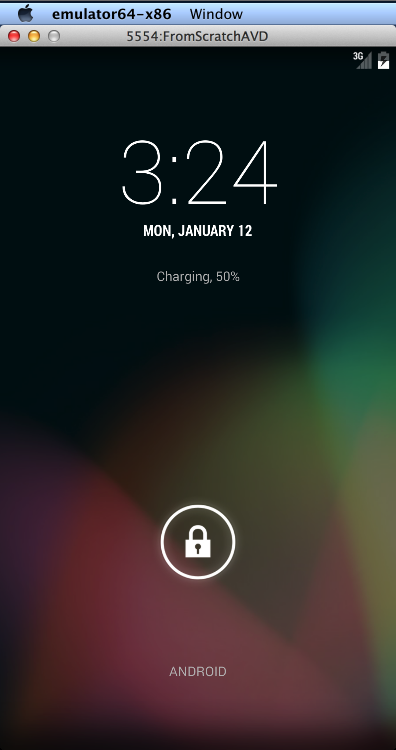


6. Next you will need an Android Virtual Device (AVD):

A. Go to Eclipse > Window >Android Virtual Device Manager and click on “Create,” to create the AVD with a target of 4.4.2.

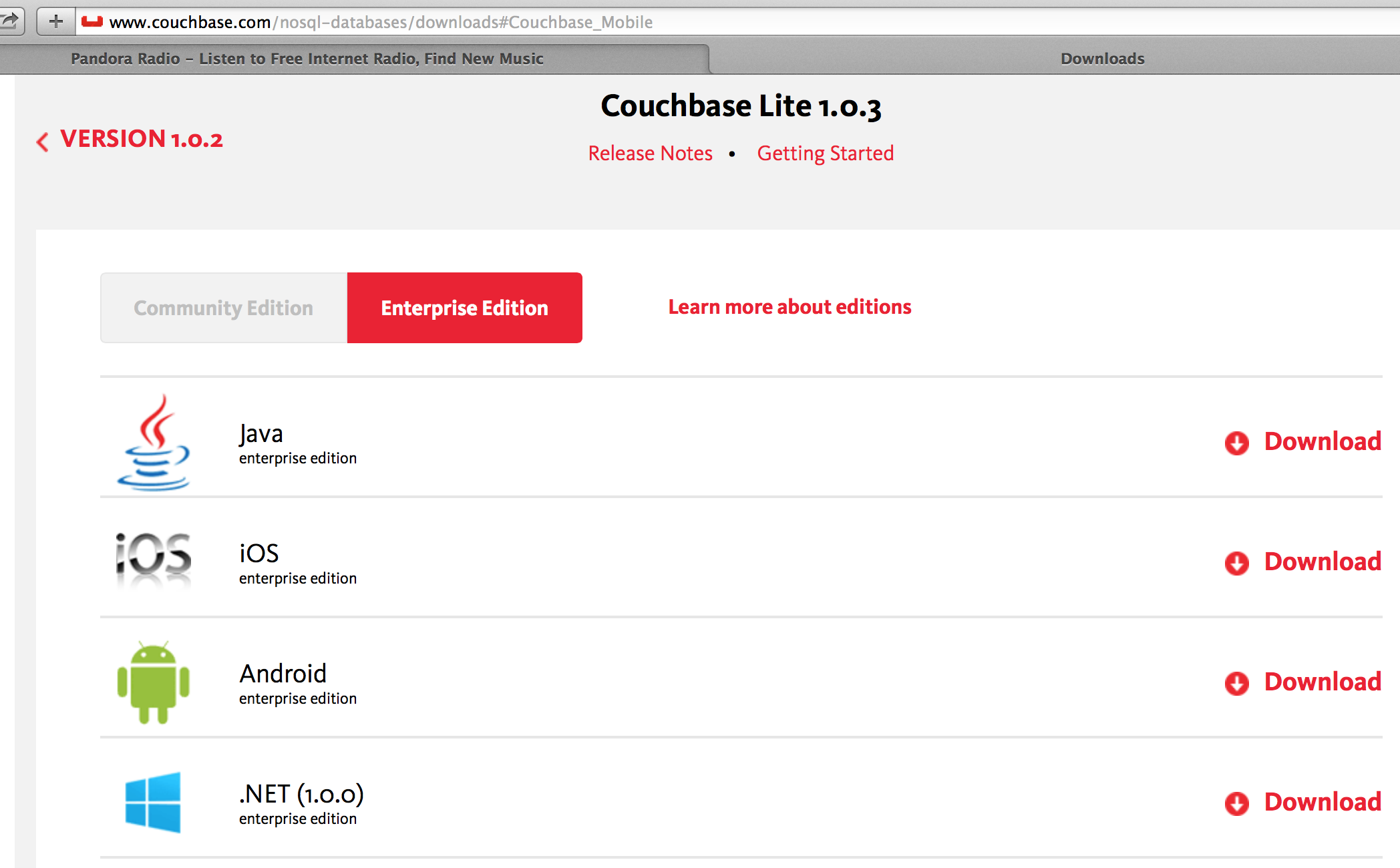


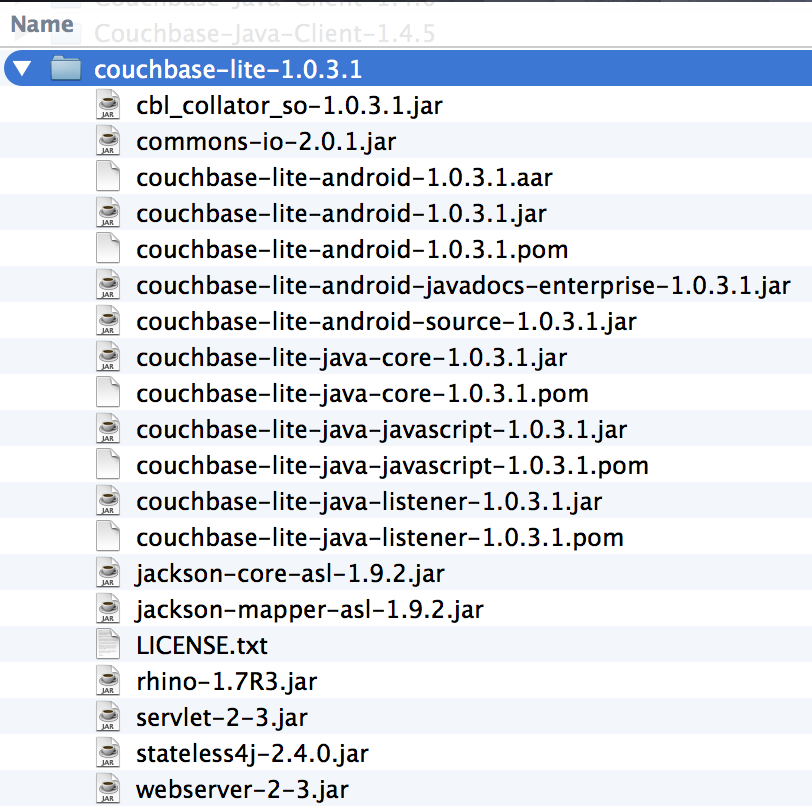
B. Click OK and then Start the AVD, which should run as something like this on your desktop:



B. Next we need to grab the Couchbase Lite SDK.

1. Download Couchbase Lite v. 1.0.3.1 or more recent: navigate to: <http://www.couchbase.com/nosql-databases/downloads#Couchbase_Mobile>

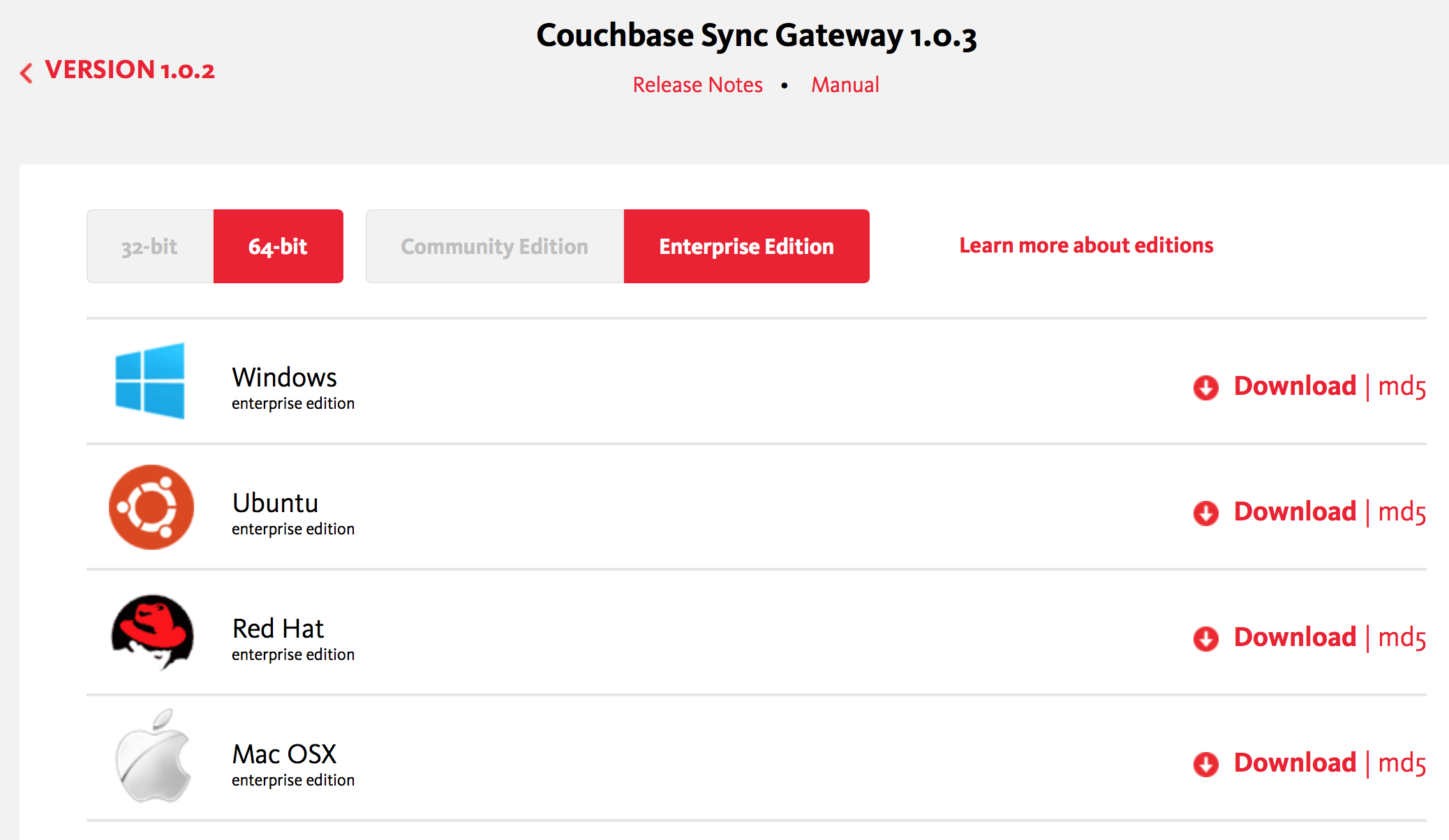




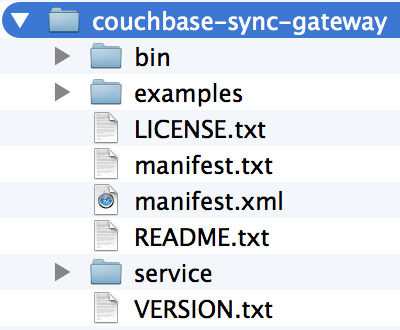
C. Download the Sync Gateway v. 1.0.3. or more recent:

1. Navigate to:

<http://www.couchbase.com/nosql-databases/downloads#Couchbase_Mobile>



2. The download file once extracted will contain:

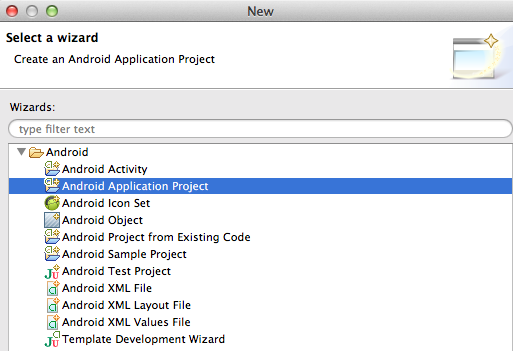


We will be using the Sync Gateway later in the tutorial, so stand by for further instructions, now that you have it downloaded.

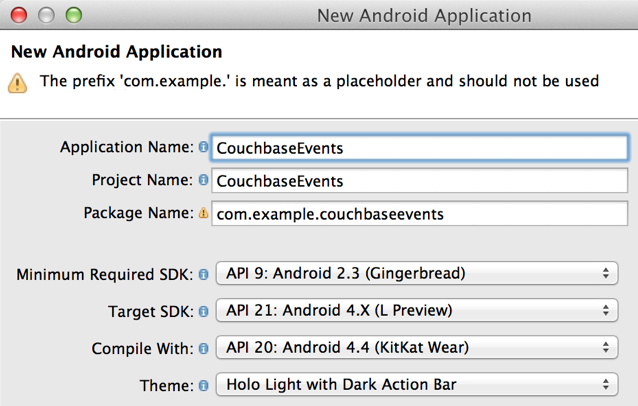
II. Development Environment.

Okay great, we are almost ready to get started with application development, we just need a project to work with.

1. In Eclipse we will need to create an Android Project, so go to Eclipse > File > New > Other > and navigate to the Android section:

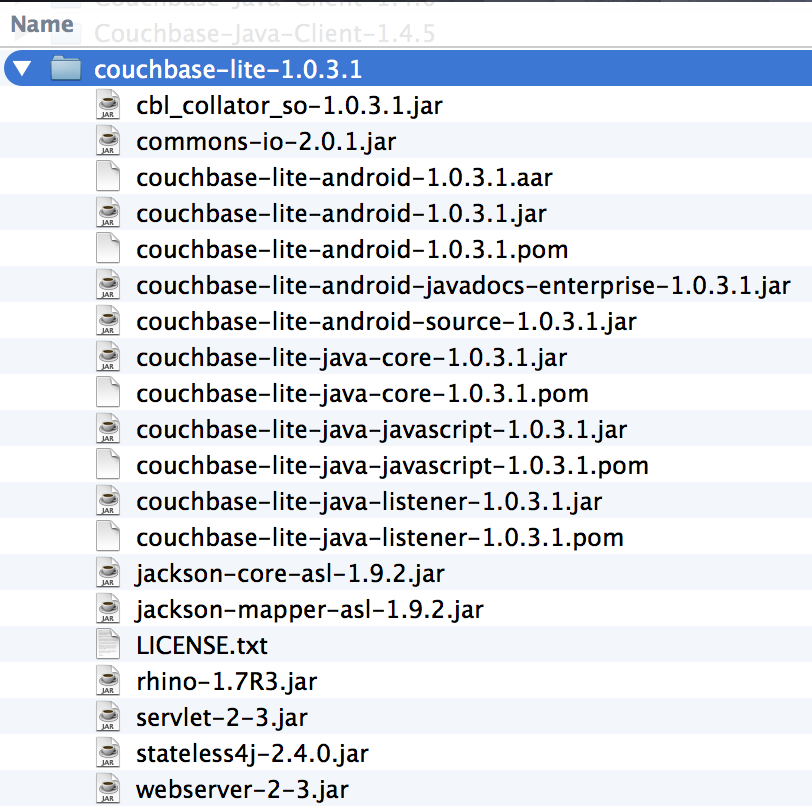


1. Select “Android Application Project” and give the project the following properties:
   1. Application Name: CouchbaseEvents
   2. Project Name: CouchbaseEvents
   3. Package Name: com.couchbase.training.couchbaseevents
   4. Change the minimum required SDK to “API 9.”
   5. Leave the other fields as the default values.

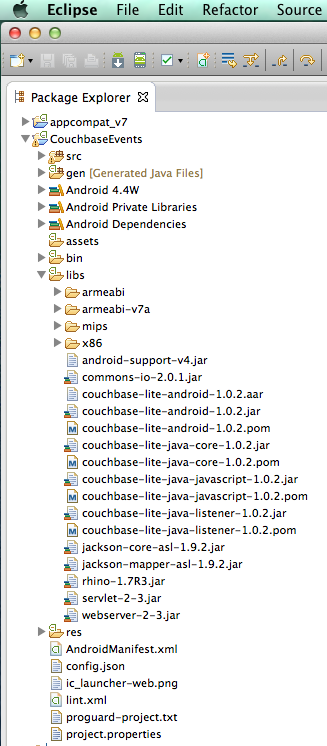
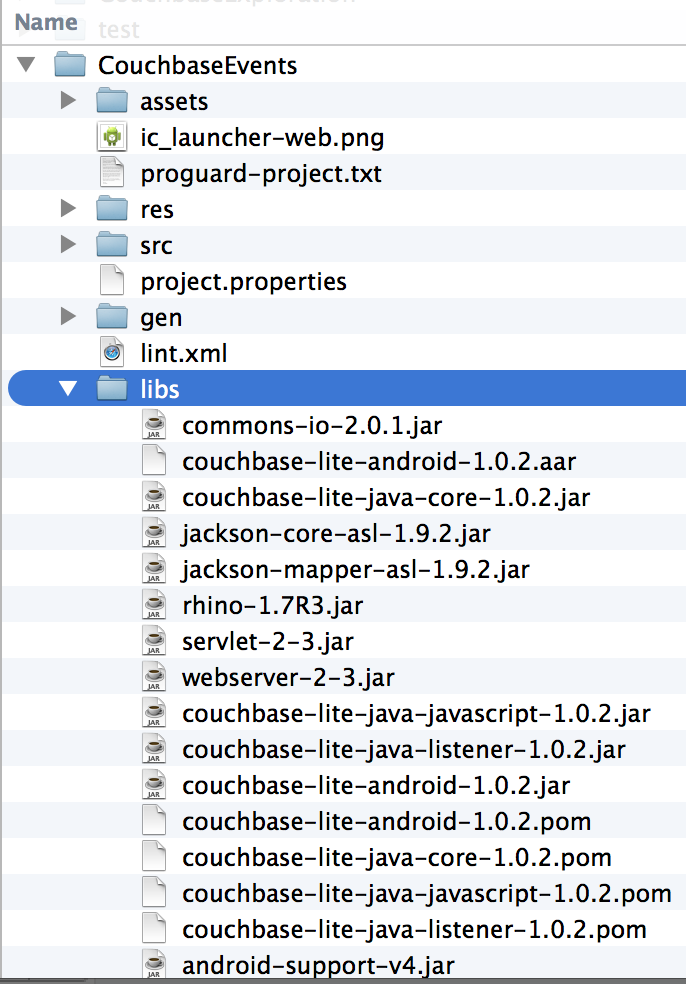
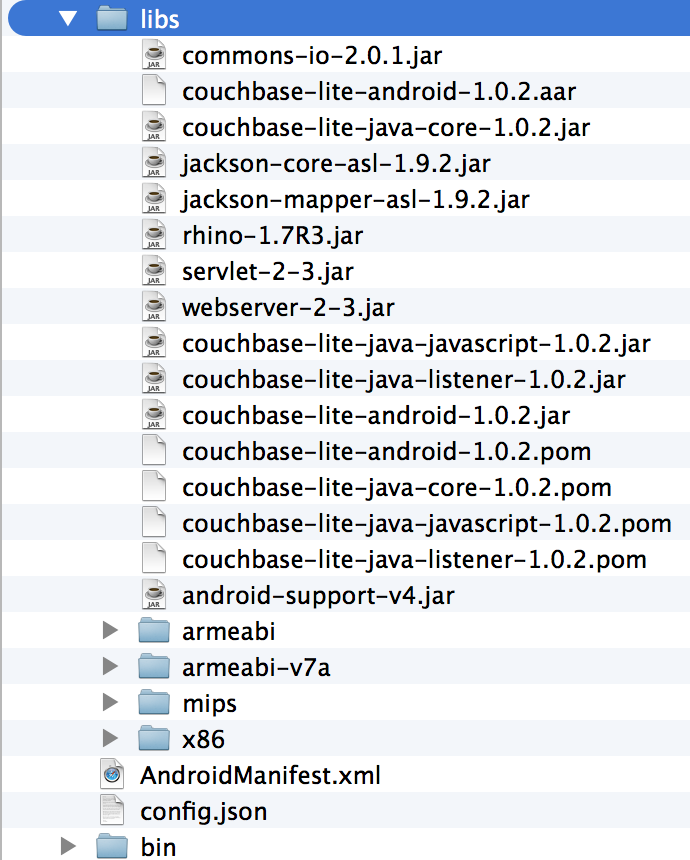


1. Click “Next” repeatedly on the next set of dialogues which are all default setting and don’t need to be changed, until you reach the “Finish” dialogue.

D. Now that the Eclipse Project is in place, let’s go back and configure the Project to work with Couchbase Lite.



E. Change the name of “cbl\_collator\_so­<versionNumber>.jar” to “cbl\_collator\_so­<versionNumber>.zip” and decompress “cbl\_collator\_so­<versionNumber>.zip” to a temporary directory. The screen shots below show the version number used at the time of the creation of this tutorial. Use the version that you have downloaded.

1. Go to that temporary directory (where you just decompressed cbl\_collator\_so­<versionNumber>.zip, and move all of the directories under “lib” to the same directory where you decompressed the Couchbase Lite for Android download file.
2. The directories under lib are named: “armeabi, armeabi­v7a, mips,” and “x86.”
3. Copy all of the directories and JAR files that came from the Couchbase Lite for Android download to the Eclipse “/workspace/CouchbaseEvents/libs” directory, which will result in your Eclipse project looking like this:
4. 
5. And your filesystem will look something like this:
6. 
7. 

III. Application Development. Writing your first Android application with Couchbase Lite.

Now we are ready. We are going to create a system that allows the user to make events (such as a party, or a seminar), and update them, etc.

1. We assume that developers are already familiar with the android.app.Activity life-cycle methods:
   1. onCreate(…), onStart(), onDestroy(), onPause(), onResume(), onRestart(), and onStop() etc.
2. Before we modify the CouchbaseEvents code we provide, let’s take a look at some of the base level code. This application is based on events the user would create, such as a party, or a rock concert they wish to attend:

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();

}

1. As we can see, the onCreate(…) method is overridden to invoke our HelloCBL method, which is currently where the action is:

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);

}

1. As you can see, the whole process starts with a com.couchbase.lite.Manager, which requires an AndroidContext and ManagerOptions to be passed to the constructor.
2. Once we instantiate the Manager, we can use that reference to invoke the factory method manager.getDatabase(…) to get a com.couchbase.lite.Database which is wrapped around the database name, in this case, “couchbaseevents” for our event planning and management app.
3. A best practice here is to actually make the Manager and Database references available to the rest of your Android application as singletons. So let’s make that happen now:

**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;

}

1. Now that we have singletons in place, let’s take a closer look at the createDocument(…) method we wrote for you:

/\*\*

\* 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;

}

1. As we can see the Database object has a Document factory, so we use it.
2. Further, we see the Document has it’s own documentId already, so we fetch it, since we promise to return it later.
3. Then we use a Map to store properties that will be written to the Document.
   1. The name of the event: “Big Party.”
   2. The location of the event: “My House.”
4. The Document has a putProperties(…) method which writes the contents of the map into the Document.
   1. The document.putProperties(…) method actually saves the passed in parameter data from the Map to the Couchbase Lite database on the local device. This is a poor name, but it is like an ORM save of the document, but only locally.
   2. Replication is never invoked by this putProperties(…) method, as we will find out later when we configure the Sync Gateway and the push and pull replication methods.
5. We return the documentId so that the caller has the document ID for their records, since they created the document, they will want to know.

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. In order 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 docuementId, that is easy asking the Database to get that document by documentId.
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. We want to know where to go!
   3. Lastly we put the Map of properties back into the document as a set of properties and values with document.putProperties(…). Believe it or not, this putProperties(…) method actually saves the data to the local Couchbase Lite database on the device so it is a poorly named method.
   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.
3. We can also add a binary attachment. As a reminder, this attachment can only be read by the Couchbase Lite (CBL) API. As of this writing, there is no other way to access this attachment. Future released of the Couchbase client SDKs will address the sharing of CBL Attachment data.

/\*\*

\* 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);

}

}

1. Let’s take a closer look at our Attachment code. 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;

}

[William Resume Here ]

Documents created and saved with Couchbase Lite go to the local database on the mobile device. When saving documents and adding attachments with revision.save() remember that the data is saved on the Couchbase Lite database, unless they are pushed to Sync Gateway with push replication. We will see how to do that next.

V. Understanding the Sync Gateway.

Now that we have our mobile client working, we will want to communicate with the Couchbase Server. In order to accomplish that, we will have to establish a connection to the Sync Gateway. This connection will allow us to achieve our Replication Strategy. There will be PUSH and PULL options on the replication of data to and from the Couchbase Server. Without the Sync Gateway, the Couchbase Lite API will not be able to communicate with the Couchbase Server. It is a required middle tier, and it resides on its own server, not on a Couchbase cluster node. The code we have written thus far is really only available on the client device, in this case Android.

1. We require this Sync Gateway in order to literally synchronize the data on the mobile device with the Couchbase Server instance residing on the network cluster that has been installed.
2. In order to accomplish this, we need to install an instance of the Sync Gateway software to act as the middle-tier, replicating the data from the device(s) to the server, and changes to the data on the server, back to the device(s).
3. An important point is that Sync Gateway is for Couchbase Lite documents. Our moto for the time being is, “If Sync Gateway didn’t create the record, then Sync Gateway won’t read the record."
4. For those interested in the Go language, Sync Gateway was written in Go: <https://golang.org>
5. Okay, so let’s get the Sync Gateway server running, and then we can add replication code to our mobile application to sync it up with the server.
6. Open a terminal or command prompt and change to the directory where you extracted the Sync Gateway download.
7. Change directories to the “bin” directory.
8. Make a new directory under the bin directory called “data.”
9. Create a file in the bin directory called “config.json.”
10. Change the config.json contents to be:

{

"log":["CRUD+", "REST+", "Changes+", "Attach+"],

"databases": {

"couchbaseevents": {

"server":"walrus:data",

"sync":`

function (doc) {

channel (doc.channels);

}`,

"users": {

"GUEST": {"disabled": false,

"admin\_channels": ["\*"]}

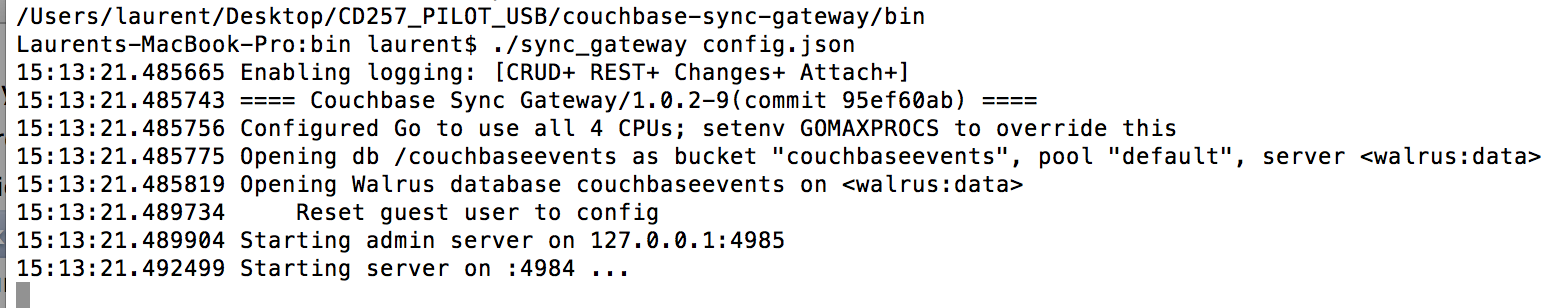
}

}

}

}

1. The file configures the Sync Gateway using Walrus as its database.
   1. Walrus is a memory­only database used for development and unit testing. It should not be used for production.
   2. Walrus resides on the Sync Gateway server, and the data does not reach the Couchbase Server instance when it is thus setup.
   3. If we want to have the data replicate all the way back to the Couchbase Server, and from the Couchbase Server to the mobile device this configuration file’s “server” child element value must change. We will show that later.
   4. If the database is Walrus, it is not Couchbase Server. And if it is Couchbase Server it is not Walrus, they are mutually exclusive.
2. The “databases” child element, gives the name of the bucket. This value needs to match the name of the database in our Mobile (CBL) code.
3. Change the directories to the bin directory: couchbase-sync-gateway/bin
4. Run this command on Linux/OS X:  ./sync\_gateway config.json



1. Once you see “Starting server on: 4984 …” your Sync Gateway is running.
   1. Important: Closing this terminal or exiting this window will require you to re-start the Sync Gateway.
2. All Mobile devices are connected to the Couchbase Server through the Sync Gateway.
3. It is possible to have the Sync Gateway present without the Couchbase server running, such as in a development environment.
4. In production, your Sync Gateway software will be installed and running on nodes outside of the Couchbase Server cluster. There are generally more than one Sync Gateway nodes.
5. Data serialization: If you are developing only using Sync Gateway (and not Couchbase server), the graceful exit of Sync Gateway serializes the in memory data held in Walrus.
6. Data Deserialization: When starting up again, the data is deserialized, and Walrus is re-populated with application data.
7. HTTPS/SSL can be used for encryption between the mobile device and Sync Gateway.
8. In fact, you can even stop short of using the Sync Gateway and store data extremely locally, on the device itself.
   1. Consider the following Use Case: ABC Company wanted to store the “Product Catalog” on the mobile device using the Couchbase Lite database not the Sync Gateway, and not Couchbase Server. This works, sped up the response times, and greatly improved the user experience.

VI. Bi-Directional Replication of Data (Push and Pull) from Couchbase Server.

Now that we have our application data on the mobile devices, and Sync Gateway, we need to carefully consider where that data should live, and how we want to incorporate new data from other mobile devices. Use Cases are naturally different, so we will not say, “you should always do…” but rather, here is how you will do what you wish. There are two main flavors of mobile replication in Couchbase: push and pull, from the perspective of the mobile device. Should we be pushing our new data to the Couchbase Server? Or should we be pulling new data from the Couchbase Server, or both? Should we be doing this continually or on an as needed basis? These are some of the many design considerations before our solution can be written.

* 1. Replication can be a one-shot, per method invocation. Or it can be continuous, whereby a separate Thread is spawned to continuously invoke the replication method for you.
  2. From the mobile device perspective, replication can be PUSHED (from our mobile device to Couchbase Server): write.
     1. The way that Couchbase Lite realizes that there is data that needs to be pushed to the Sync Gateway is the Couchbase Lite database has flags on new data, or changes, that need to be push replicated to the Sync Gateway.
  3. Replication can also be PULLED (From Couchbase Server to our mobile device): read.
     1. The way that the Sync Gateway realizes that there is new data in the Couchbase Server is that the Sync Gateway has advanced Listeners which are listening for changes in the Couchbase Server, and upon changes pulls data to the Sync Gateway.
  4. Replication optionally can be authenticated before it is allowed at all.
  5. All the invocations from the Couchbase Lite SDK are ultimately translated into REST invocations against the Sync Gateway.
  6. An important note about using Couchbase “SmartClient” SDKs in conjunction with the Couchbase Lite API:
     1. If a Java SDK Smart Client writes a document to Couchbase Server 3.0 (or 2.5) and CBL wants to read that, it will not work directly. For that solution we have different options.
     2. The simple answer for now is the Couchbase Smart client SDKs are all read only against the Couchbase Lite documents. For example, if a CBL client writes to Couchbase Server, and the Java SDK smart client wants to read that document, that will be fine, we have tested it and it works.
  7. So, now let’s see how we can get the replication code working, and round trip this data from the mobile device to the Couchbase Server and back again:
     1. First we need to configure Android with the right AndroidManifest.xml elements:
        1. Add this element to the file:

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

* + 1. Next we need to create a Sync URL for use by the Couchbase Lite API. The URL we are aiming for is: <http://10.0.2.2:4984/couchbaseevents>
    2. However, we also know that Sync Gateway allows for HTTPS/SSL connections, so we have to take that into consideration as well.
    3. Also, both the PUSH as well as the PULL replication strategies require this URL, so we can easily encapsulate this work in factory method, such as createSyncURL(…), let’s do that now:

private URL createSyncURL(boolean isEncrypted){

URL syncURL = null;

String dns = null;

/\* Example URL: "http://10.0.2.2:4984/couchbaseevents" \*/

if (isEncrypted) {

dns = "https://10.0.2.2"; //Encrypted Sync Gateway DNS

}

else {

dns = "http://10.0.2.2"; //Sync Gateway DNS

}

String port = "4984"; // Sync Gateway data port

String dbName = "couchbaseevents";

try {

syncURL = new URL(dns + ":" + port + "/" + dbName);

}catch (MalformedURLException me) {

me.printStackTrace();

}

return syncURL;

}

* 1. Now that we have our URL factory method, let’s use it to get this Replication party started!
  2. In the Couchbase Lite API, the replication strategies are abstracted by a Replication object.
  3. The Database object has Replication factory methods for both PUSH and PULL. Let’s see how they work, starting with PULL (reading from the server):

private void startPullReplication()

throws CouchbaseLiteException {

Replication pull = this.getDatabaseInstance()

.createPullReplication(this.createSyncURL(false));

/\* actually causes a Thread to be instantiated \*/

pull.setContinuous(true); //false is default

/\* setContinuous(true) was invoked so

\* it starts a Thread for continuous replication\*/

pull.start();

}

1. Let’s take a closer look at our code. We start by invoking our Database instance Replication factory method createPullReplication(…) and passing over our Sync URL.
2. Now that we have a Replication object, of type PULL, we can start it. We have a choice to make it a one shot replication or continuous replication:
   1. One shot replication means that setContinuous(false) is present.
   2. Continuous replication is only possible by setContinuous(true) and then starting that thread with the start() method.
3. The PUSH Replication is identical except that the method createPushReplication(…) is invoked instead. The rest of the code is the same so we won’t show that here.
4. It is possible to add authentication to the Sync Gateway, we will cover that later.

VII. Adding Sync Gateway Authorization.

Many Use Cases require an authentication and authorization policy on data replication. Sync Gateway provides a mechanism for this.

* 1. We can add a security layer to the Sync Gateway, both at the server layer and within our code.
  2. In order to accomplish this we need to take a number of steps, beginning with stopping the Sync Gateway server, naturally.
  3. Then we need to configure our security. Locate the config.json file and modify its contents to read:

{

"log":["CRUD+", "REST+", "Changes+", "Attach+"],

"databases": {

"couchbaseevents": {

"server":"walrus:data",

"sync":`

function (doc) {

channel (doc.channels);

}`,

"users": {

"GUEST": {"disabled": **true**,

"admin\_channels": ["\*"]}

}

}

}

}

1. Let’s notice that the change here is the GUEST disabled value changes to true, this will force authentication and disallow guest access.
2. Now all applications wanting authorization to use the Sync Gateway must first authenticate.
3. Restart the Sync Gateway as per the previous instructions in the section on Sync Gateway.
4. Restart your Android application and notice attempts to use the Sync Gateway should be throwing errors such as:

“Unauthorized… HTTP:#004: ­­>401Loginrequired (0.0ms)”

1. It’s good that we have errors, this means the security is working! Now we will need to add an authorized user to the Sync Gateway and our codebase as well.
   1. We will use the Sync Gateway REST Admin interface (Port #4985) to add a user. Let’s do this now.
   2. Open the terminal window and issue the following curl command:

curl ­X POST http://127.0.0.1:4985/couchbaseevents/\_user/ ­d '{"name":"laurent8","password":"meher056"}' ­H "Content­Type: application/json"

* 1. That command created a new user with the username “laurent8” and the password “meher056”. This is the same username and password you will need to authenticate with in your Android code using the API com.couchbase.lite.auth.Authenticator.
  2. Let’s make sure the PUSH and PULL Replication strategy that we created are now able to authenticate.
  3. Add the following code to your android.app.Activity onCreate(…) method:
     1. authenticator = AuthenticatorFactory.

createBasicAuthenticator("laurent8", "meher056");

* 1. Add an instance variable to hold the Authenticator itself:
     1. Authenticator authenticator = null;
  2. Modify your replication methods, before invoking the pull.start() method, invoke setAuthenticator(…):
     1. pull.setAuthenticator(authenticator);
  3. Perform the same step for the push replication method.
  4. Now both pull and push replication should be working securely.

VIII. Working with Couchbase Lite Views.

In Couchbase Server we have the notion of a View (or Index) created from a subset of Document data. CBL also allows for View creation, however the View API is significantly different, and the resulting views are also stored differently. Let’s find out how we can create and leverage views in CBL.

1. A View created in Couchbase Lite is NOT maintained as a View in the Couchbase Server at all (see: <http://developer.couchbase.com/mobile/develop/guides/couchbase-lite/native-api/view/index.html>)
   1. In Couchbase Lite, the View only exists on the mobile device.
   2. A View is not a Query. There is a separate CBL Query API as we will soon understand.
2. We will use a com.couchbase.lite.View for our work here.
3. There is a factory method on the Database called getView(“name”) for creating this object. We must provide the database a unique name for each view. Let’s encapsulate this work now for our application development:

**private** View getView(String name) {

View view = **null**;

**try** {

view = **this**.getDatabaseInstance().getView(name);

}

**catch** (CouchbaseLiteException cble) {

cble.printStackTrace();

}

**return** view;

}

1. We simply pass the parameter name to the factory method and return the freshly created View.
2. Now that we have this method, we can write a specialized method-per-View for each View we wish to create.
3. For example if we want to create a specialized view for eventsByDate we would write this method:

**public** **void** createEventsByDateView() {

View eventsByDateView = **this**.getView("eventsByDate");

eventsByDateView.setMap(

**new** Mapper(){

@Override

**public** **void** map(Map<String, Object> document,

Emitter emitter) {

/\* Emit data to matieralized view \*/

emitter.emit(

(String) document.get("date"), **null**);

}

},

"1" /\* The version number of the mapper... \*/

);/\* end setMap(…) invocation whew! \*/

/\* Test it out... \*/

printQueryToLog(eventsByDateView);

}

1. Let’s discuss this method, first we leverage our private getView(…) method we just wrote to fetch a fresh View from the Database, with the name “eventsByDate.”
2. Now we have to set the map function on the view, using setMap(…)
   1. We do this dynamically with a new Mapper object passed in anonymously on the fly.
   2. This works because of the @Override of the map(…) method, where all the action is.
   3. We also provide a version number “1” of the map code.
3. The map method provides the references to the document we wish to index, and the emitter which will emit that data to the index.
4. We are only interested in the Event date at this point, so we emit that date and null.
5. Lastly we test our work in the printQueryToLog(…) method. Let’s take a look at that code now:

private void printQueryToLog(View view) {

// Get instance of Query from factory…

Query orderedQuery = view.createQuery();

orderedQuery.setDescending(true);

orderedQuery.setStartKey("2015");

orderedQuery.setEndKey("2014");

orderedQuery.setLimit(20);

try {

QueryEnumerator results = orderedQuery.run();

/\* Iterate through the rows to get the document ids \*/

for (Iterator<QueryRow> it = results; it.hasNext();) {

QueryRow row = it.next();

String docId = (String) row.getValue();

Event event = eventRepository.getById(docId);

Log.i(CBLSingleton.TAG, "Found party:" + event);

}

} catch (CouchbaseLiteException e) {

Log.e(CBLSingleton.TAG, "Error querying view.", e);

}

}

1. Let’s take a closer look now at this method to print the results of our Query to the log.
   1. First we need a Query object which we can get from the View’s factory method. See documentation here: http://developer.couchbase.com/mobile/develop/guides/couchbase-lite/native-api/query/index.html
   2. We want the most-future events first, and the oldest most-past events last, so we modify our Query setDescending(true).
   3. We don’t intend to look further in the future than 2015, and we don’t care about events prior to 2014, so we set start and end keys.
   4. We only want the first 20 results, so we set the limit to 20, which is the maximum number of documents to return.
   5. After a Query object is set up properly, you call its run method to get the results.
      1. The results are returned as a QueryEnumerator object, which mainly serves as an enumerable collection of QueryRow objects.
   6. We then iterate over the results getting each QueryRow.
   7. For each QueryRow in the results, we find the docId and use an EventRepository to fetch the actual complete Event object from the Sync Gateway.

IX. Understanding a LiveQuery.

To help us track the changes that could be occurring in such a mobile development environment, the CBL has an implementation of the Observer pattern known as a LiveQuery. We will use the LiveQuery to go deeper into the possibilities of how listening for changes can help us provide solutions.

1. We have the concept of a “live query” which is encapsulated by the class: com.couchbase.lite.LiveQuery.
2. The meaning of a LiveQuery is that it is a type of Observer [GoF] on the Query object. The Observer pattern sets up a relationship between two objects whereby the Observer (or listener) is notified of some event by the object it is observing. The listener must register with the object to which it is listening, using some method, so that the object knows who is registered and listening. Then the listener is “notified” of changes by having a method invoked by the object to which it is listening.
3. We get a LiveQuery from the Query factory method toLiveQuery().
4. Because the LiveQuery came from the Query itself, it is already set to Observe the Query data. That is, the Query contains data from the View. We are going to listen (observe) that data, using a ChangeListener.
5. We have to perform the following steps to accomplish this, and then we can be notified per the Observer pattern of any changes to the data underlying the Query we are Observing:
6. Invoke the LiveQuery addChangeListener(…) method and pass as a parameter a new ChangeListener. This registers our listener.
7. This ChangeListener should override the method changed(…)
8. Start the LiveQuery so it is listening for changes to the data bounded by the Query.
9. Let’s take a look at the code for this now:

LiveQuery myLiveQuyery = eventsByDateView.createQuery().toLiveQuery();

myLiveQuery.addChangeListener(

/\* instantiate the ChangeListener right here \*/

new LiveQuery.ChangeListener() {

@Override /\* A notification listener method \*/

public void changed(ChangeEvent changeEvent) {

Log.i(“couchbaseevents”,

"Data changed in the Query…");

}

}

);

myLiveQuery.start(); //Start listening for changes...

1. Let’s understand this listener code, and its deeper ramifications.
2. First we got our LiveQuery from the factory.
3. Next we add a ChangeListener to the LiveQuery, so we can do something, whenever data in our Query changes.
4. We have to instantiate one, and override the appropriate method: changed()
5. Within the method we write to the log file. Naturally you could do whatever you want in this method.
6. Let’s understand more about the incoming ChangeEvent, and what it means that the changed() method was invoked, and given a reference to this event by the CBL API.
7. The LiveQuery is listening for changes to Document versions.
8. Let’s say that DocumentA is part of the Query.
9. When the revision of DocumentA is saved after an update, there is a new version of DocumentA, and that new version being available for read causes the ChangeEvent to be instantiated and passed over to the changed(…) ,method.
   1. This means that every version of the document causes a new ChangeEvent to come over.
   2. This means that if DocumentA had one data element updated (changed), such as “address,” it was cause a ChangeEvent to be fired.
   3. This also means that if DocumentA had three data elements change, such as “address,” “date” and “description” still only one ChangeEvent would be fired off, because the LiveQuery only listens at the Document version level, not at the data attribute level within the document.
   4. Furthermore, we receive the ChangeEvent but no details about what within the document was changed.
   5. In order to figure out what exactly has changed, we would need to run a new Query, and compare the results to that of the previous query. More likely, some UI element value has entirely changed.
   6. Therefore we say, the LiveQuery notifies the ChangeListener THAT data in the Query has been updated, not HOW that data has been updated.
   7. To recap, the ChangeListener changed(…) method is invoked each time any document in the Query has a version change (updated version), and passed a ChangeEvent.
   8. The ChangeEvent itself has access to the original Query data (before any change occurred) via the getRows() method which provides a QueryEnumerator over that original data, like this:

public void changed (ChangeEvent changeEvent) {

QueryEnumerator originalData = changeEvent.getRows();

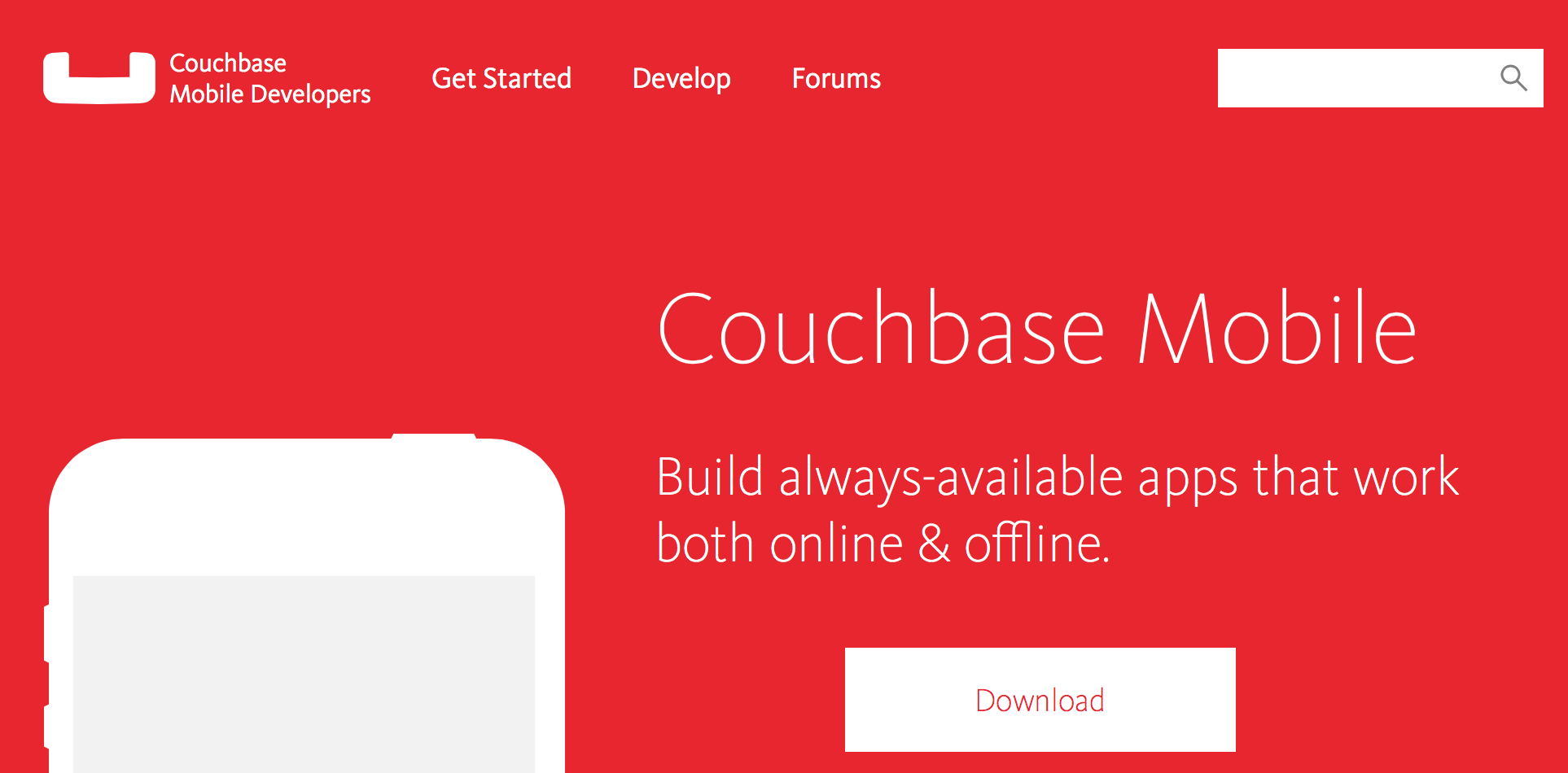
/\* work with Query data \*/

}

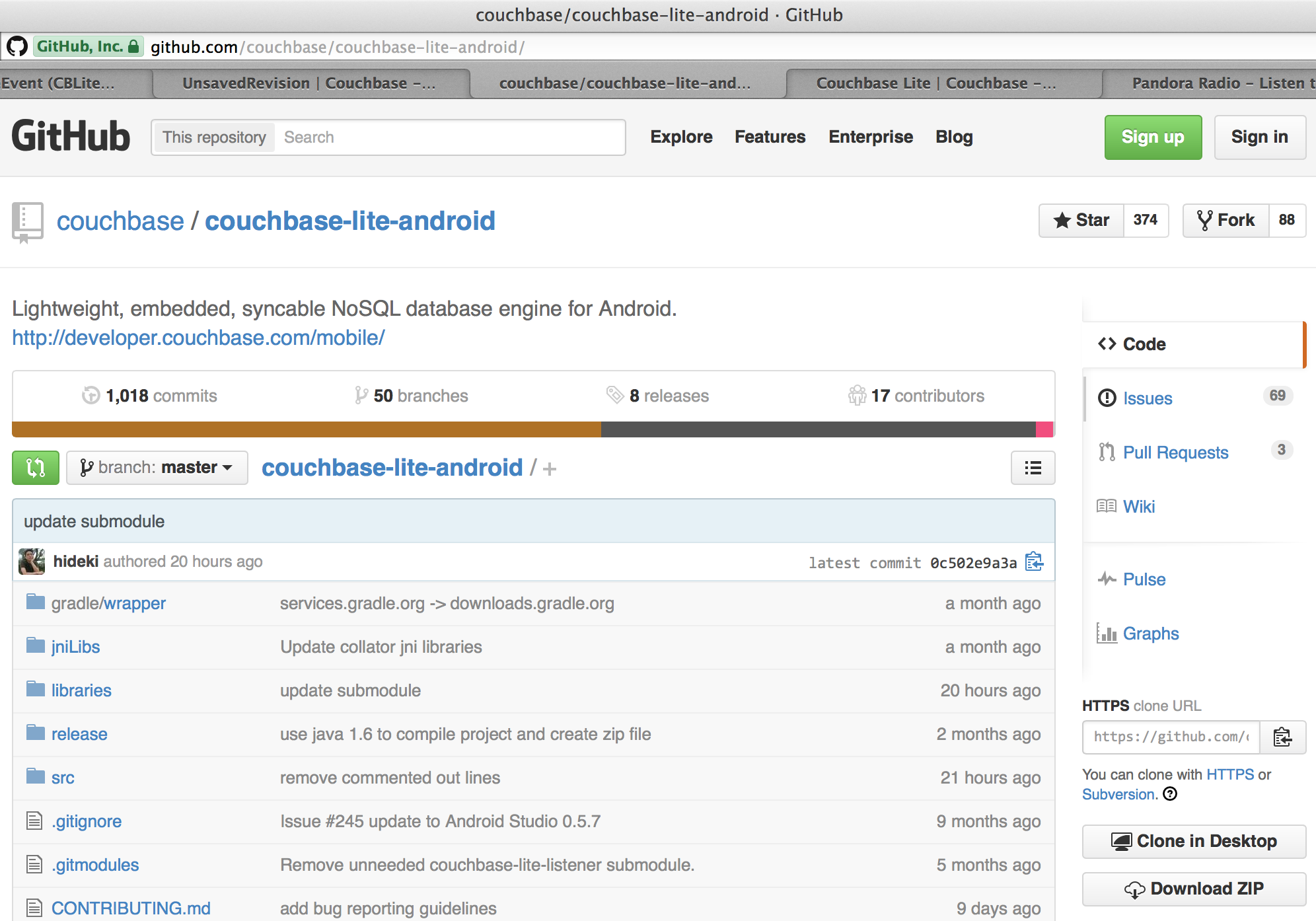
X. Some Couchbase Lite Online Resources:

The Couchbase Lite world is being updated and evolving all the time. To this end we provide you with many resources to ensure your success as you build your mobile applications with Couchbase:

1. Visit the main developer site: <http://developer.couchbase.com/mobile/>

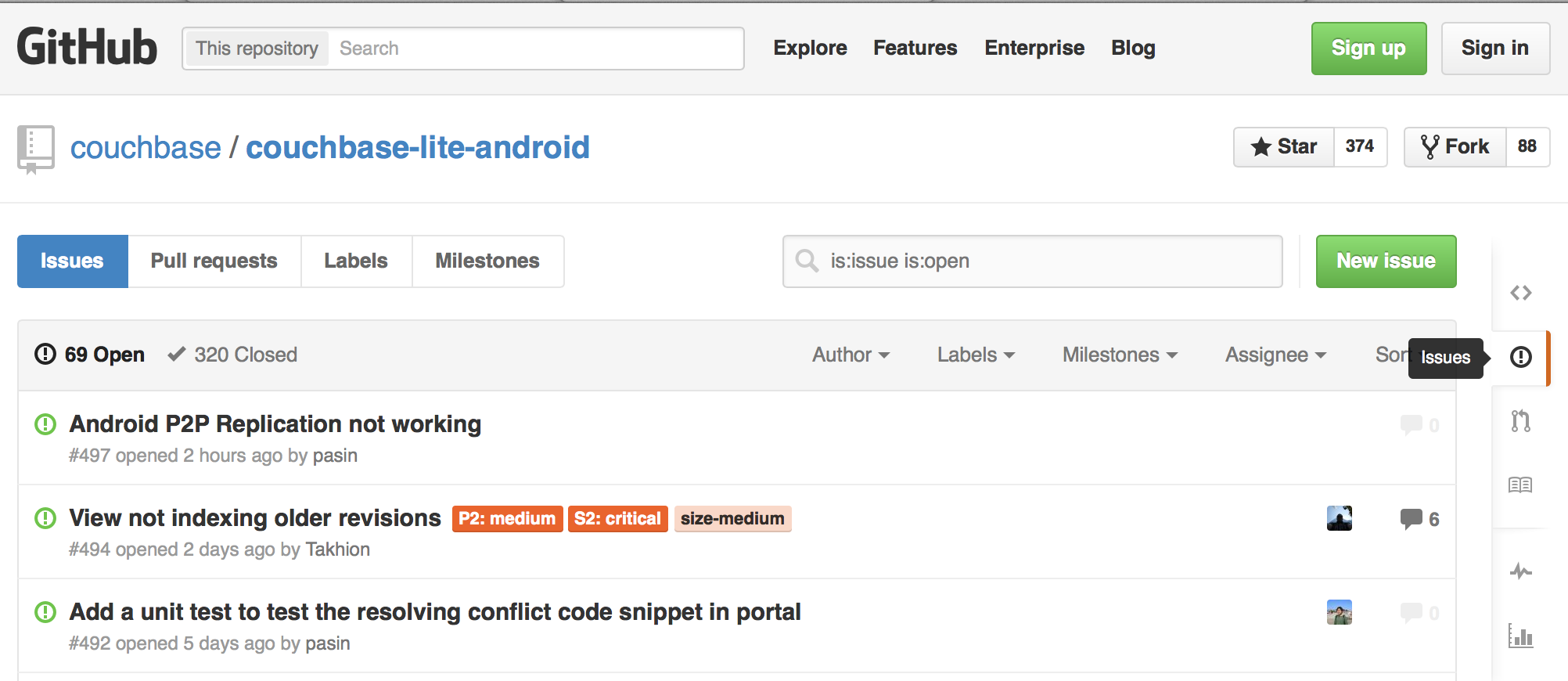


2. Review the main page GitHub site for Android: <https://github.com/couchbase/couchbase-lite-android/>

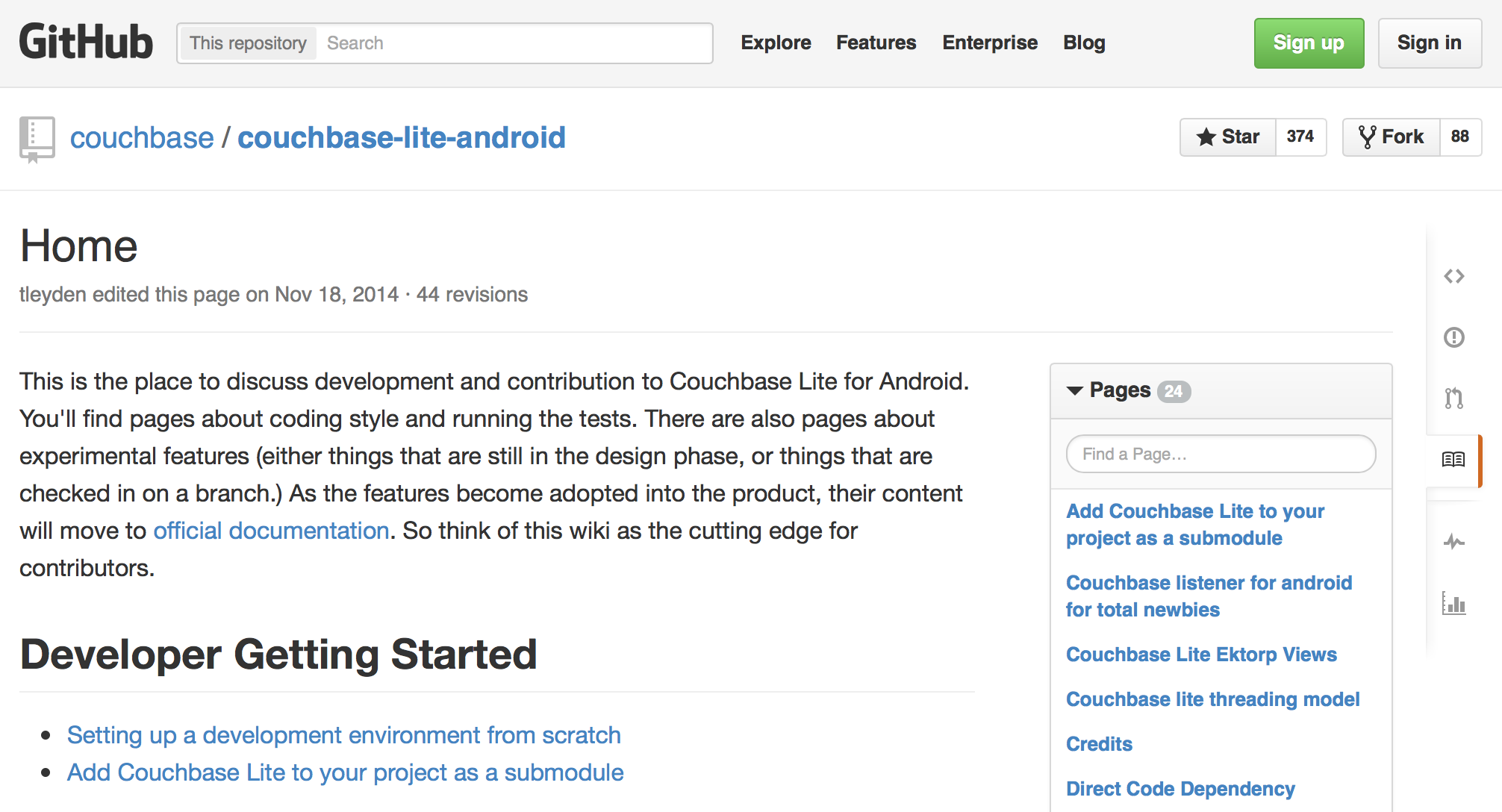


3. Click on the issues tab on the top right of the page and review the known

issues:



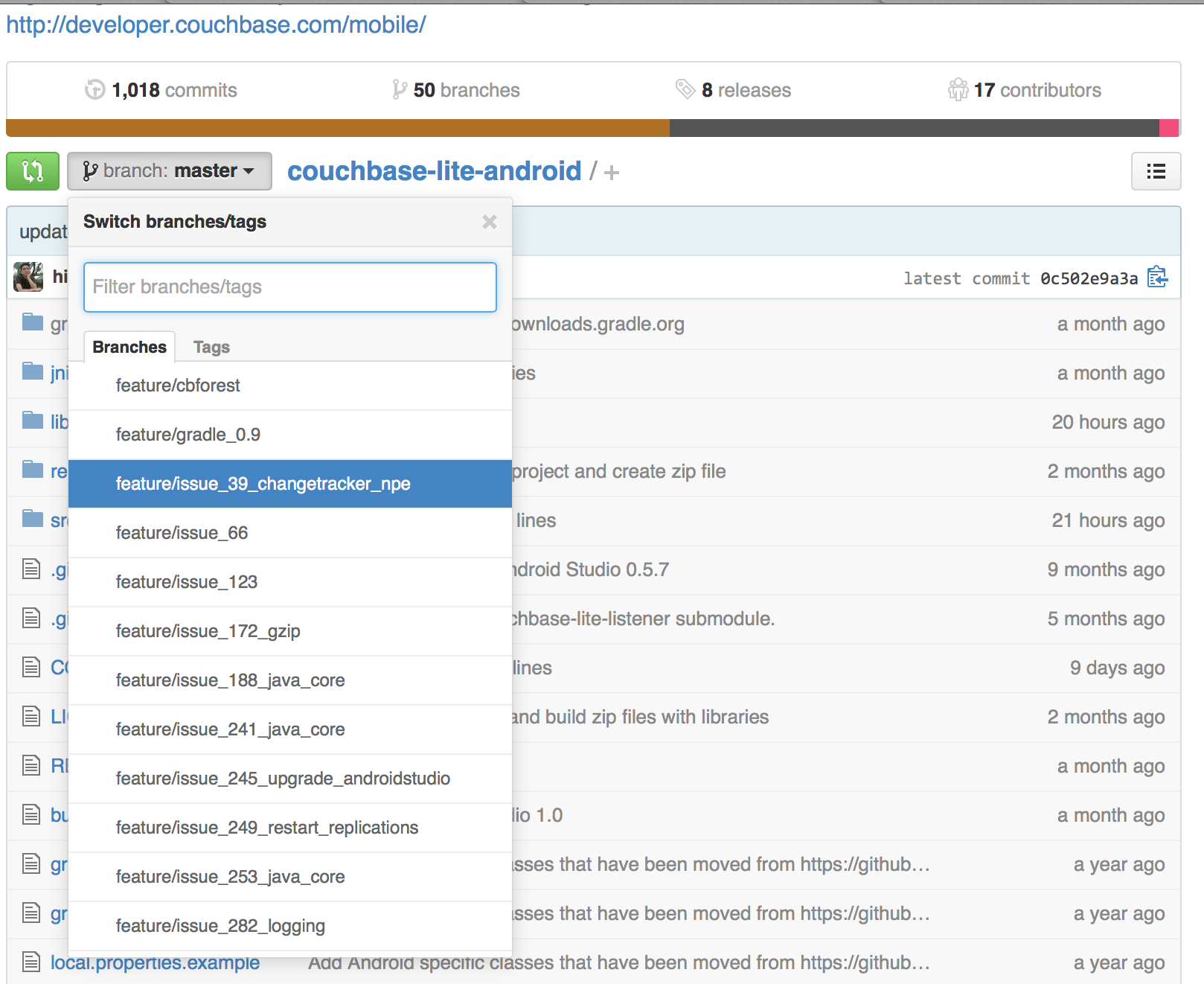
4. Click on the “Wiki” link on the right navigation.



These pages provide help on advanced topics that we don’t cover in this tutorial.

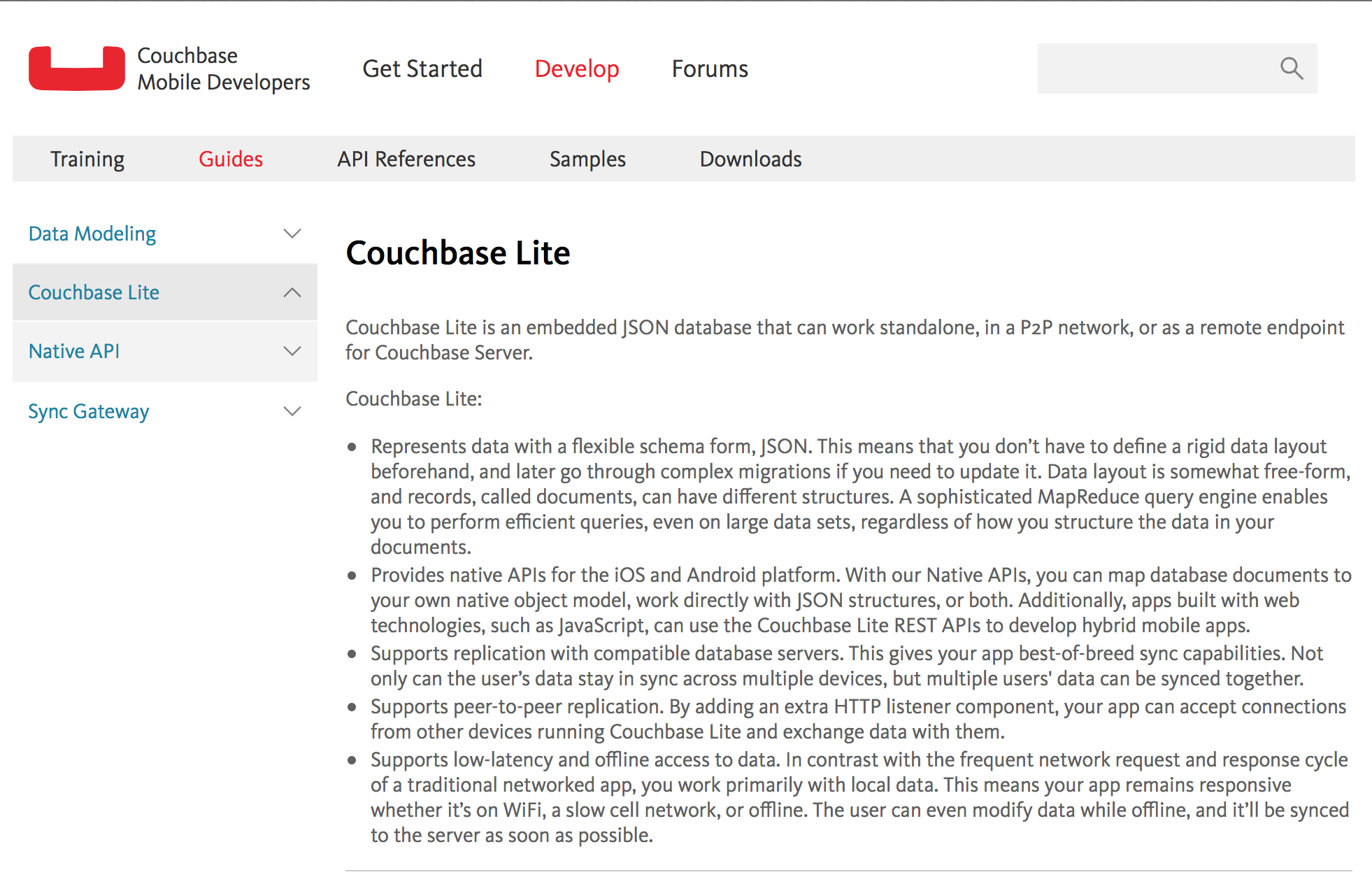
5. Click on the branches drop down and look through the releases. You can find

the latest release of Couchbase Lite for your platform:



7. Review the Couchbase Lite API guide site:

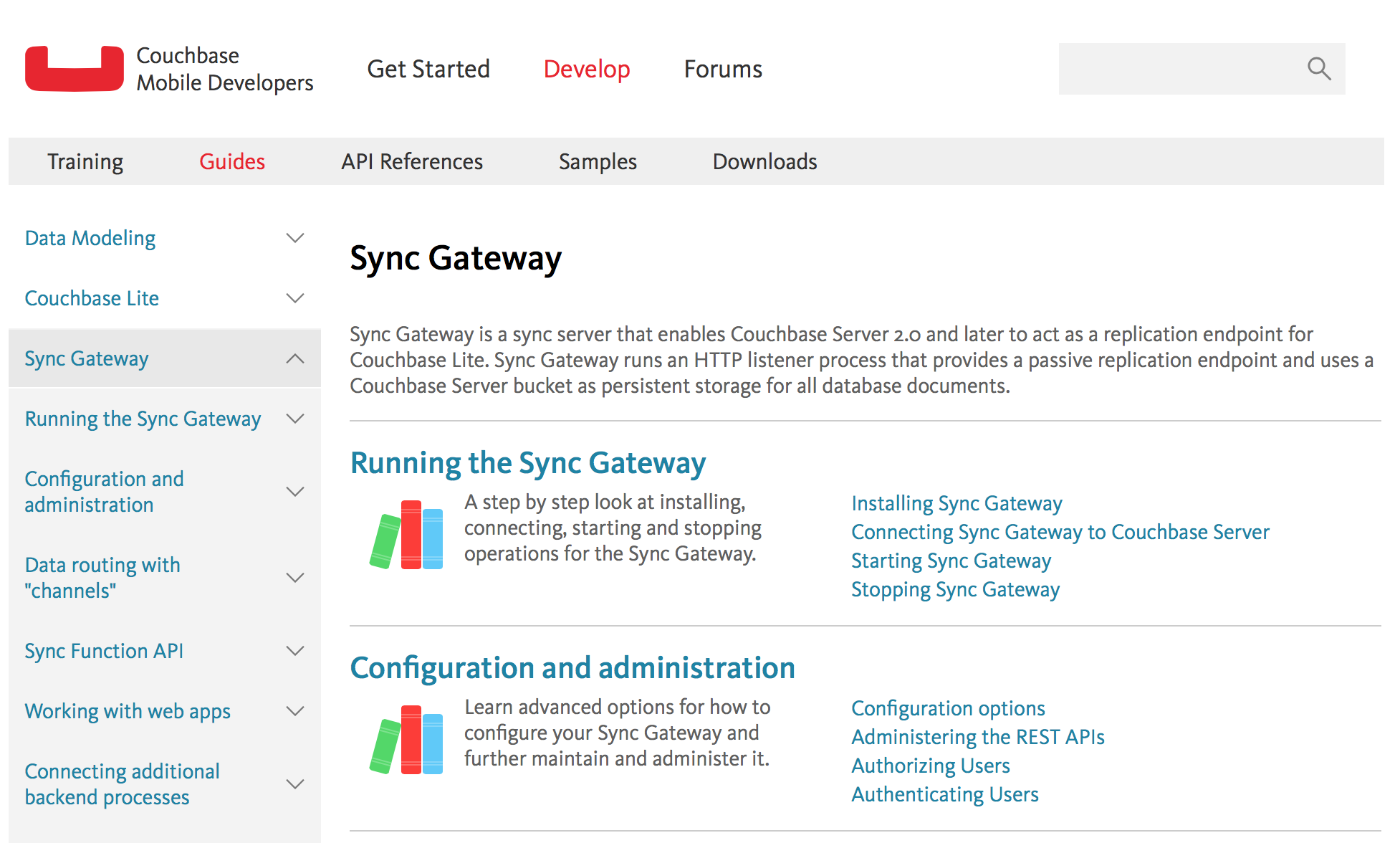
<http://developer.couchbase.com/mobile/develop/guides/couchbase-lite/index.html>



This site provides information and code samples for the API.

8. Review the Sync Gateway guide site:

http://developer.couchbase.com/mobile/develop/guides/sync­gateway/index.ht ml



This site provides information and code samples for the API.

9. Couchbase Lite can be cloned from GitHub and built from that source code. You can use this instead of the releases on the main developer site. The instructions for Android can be found on the main page under the ‘Building Couchbase Lite on command line via gradle’ section:



Thank you for working with our Couchbase Lite tutorial for Android developers.

For more information contact:

Laurent Weichberger

training@couchbase.com