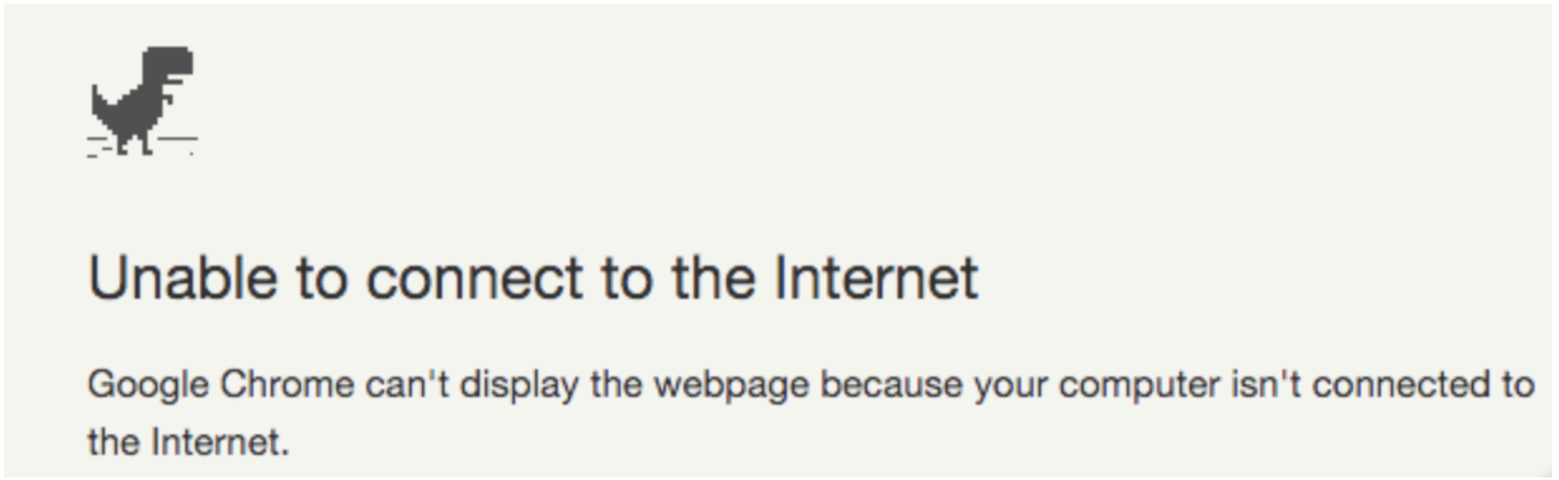
Lecture #6
Synchronizing data

Mobile Applications 2018-2019

Offline Mode



- User needs access to his data.
  - Spotty Internet connection.
  - No connection at all.
- Acting like a cache.

Issues

- Stale/old data.

Issues

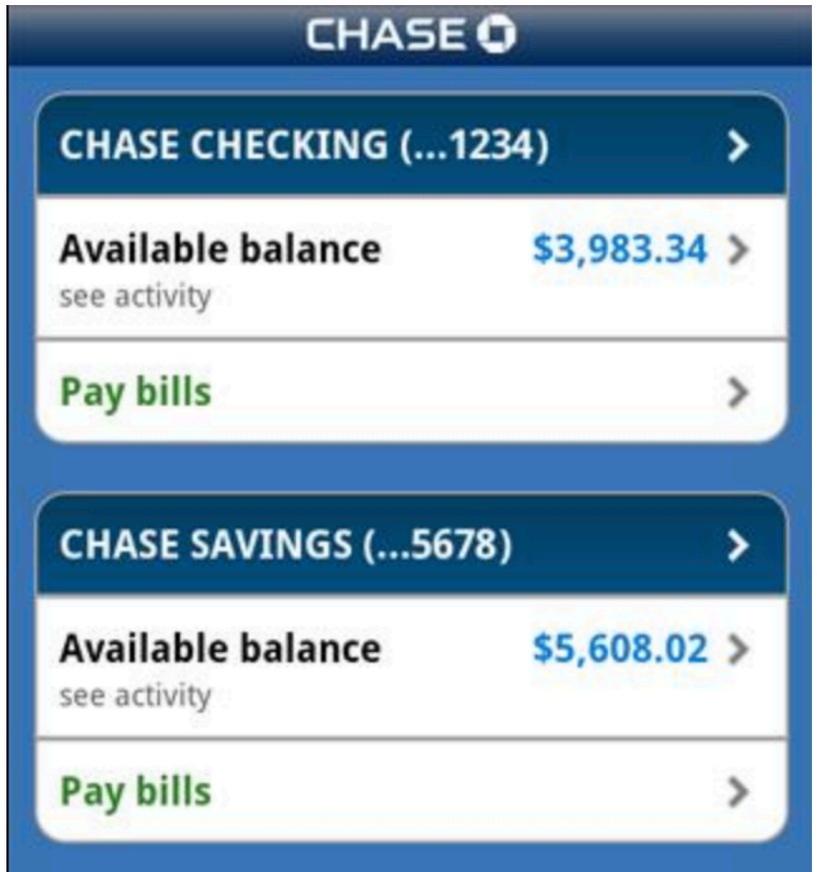
- Stale/old data.

Issues

- Stale/old data.

Issues

- Stale/old data.
- Inconsistent data.



Good morning! You have **a card payment due** and more...

Accounts | Pay & transfer | Account management

BUSINESS CARD (...6789)

ink.

$7,285.91
Current balance

CARDS (3)

Ultimate Rewards®
Use your points

13,000
Total points

BUSINESS CARD (...6789)

We didn't receive last month's payment of $67.08

We've included this amount in your new minimum payment,

credit

.00

a balance >

payment due

Total

$15

Bala

$7,2

CHASE

CHASE CHECKING (...1234) >

Available balance $3,983.34 >
see activity

Pay bills >

CHASE SAVINGS (...5678) >

Available balance $5,608.02 >
see activity

Pay bills >

Good morning! You have **a card payment due** and more...

Accounts Pay & transfer Account management

BUSINESS CARD (...6789)

ink.

$7,285.91
Current balance

CARDS (3)

Ultimate Rewards®
Use your points

13,000
Total points

BUSINESS CARD (...6789)

We didn't receive last month's payment of $67.08

We've included this amount in your new minimum payment,

credit
.00
a balance >
n payment due
Total
$15
Balan
$7,2

CHASE

CHASE CHECKING (...1234) >

Available balance $3,983.34 >
see activity

Pay bills >

CHASE SAVINGS (...5678) >

Available balance $5,608.02 >
see activity

Pay bills >

Issues

- Stale/old data.
- Inconsistent data.
- Bad data.

Issues

- Stale/old data.
- Inconsistent data.
- Bad data.

Issues

- Stale/old data.
- Inconsistent data.
- Bad data.

Wrong/Costly/Embarrassing decisions!

Possible Causes

Possible Causes

- Delays.

Possible Causes

- Delays.
- Misconfigurations.

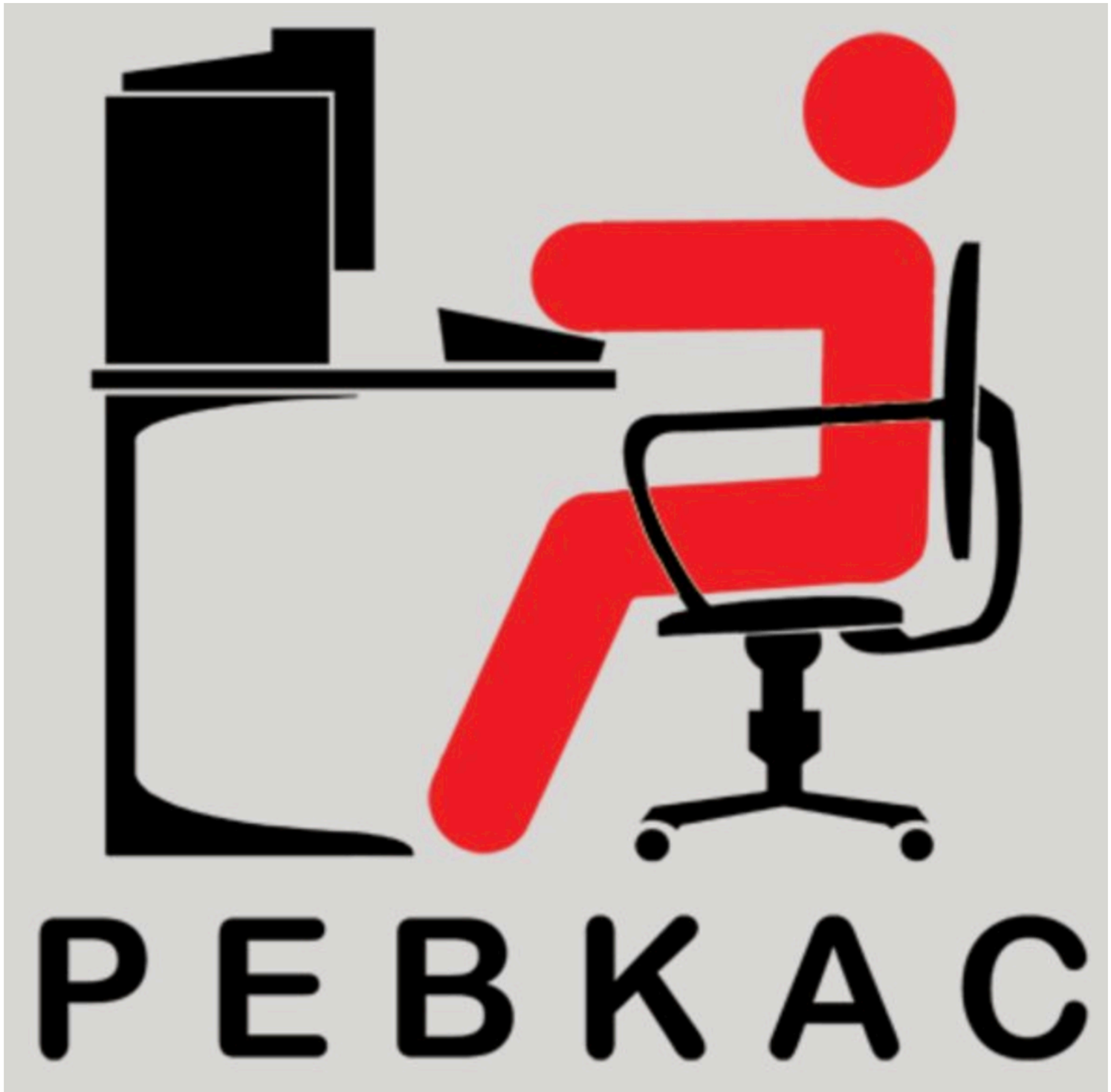
Possible Causes

- Delays.
- Misconfigurations.
- Insufficient resources.

Possible Causes

- Delays.
- Misconfigurations.
- Insufficient resources.
- Other/All combined.

Most of the time



Most of the time

Synchronizing Data

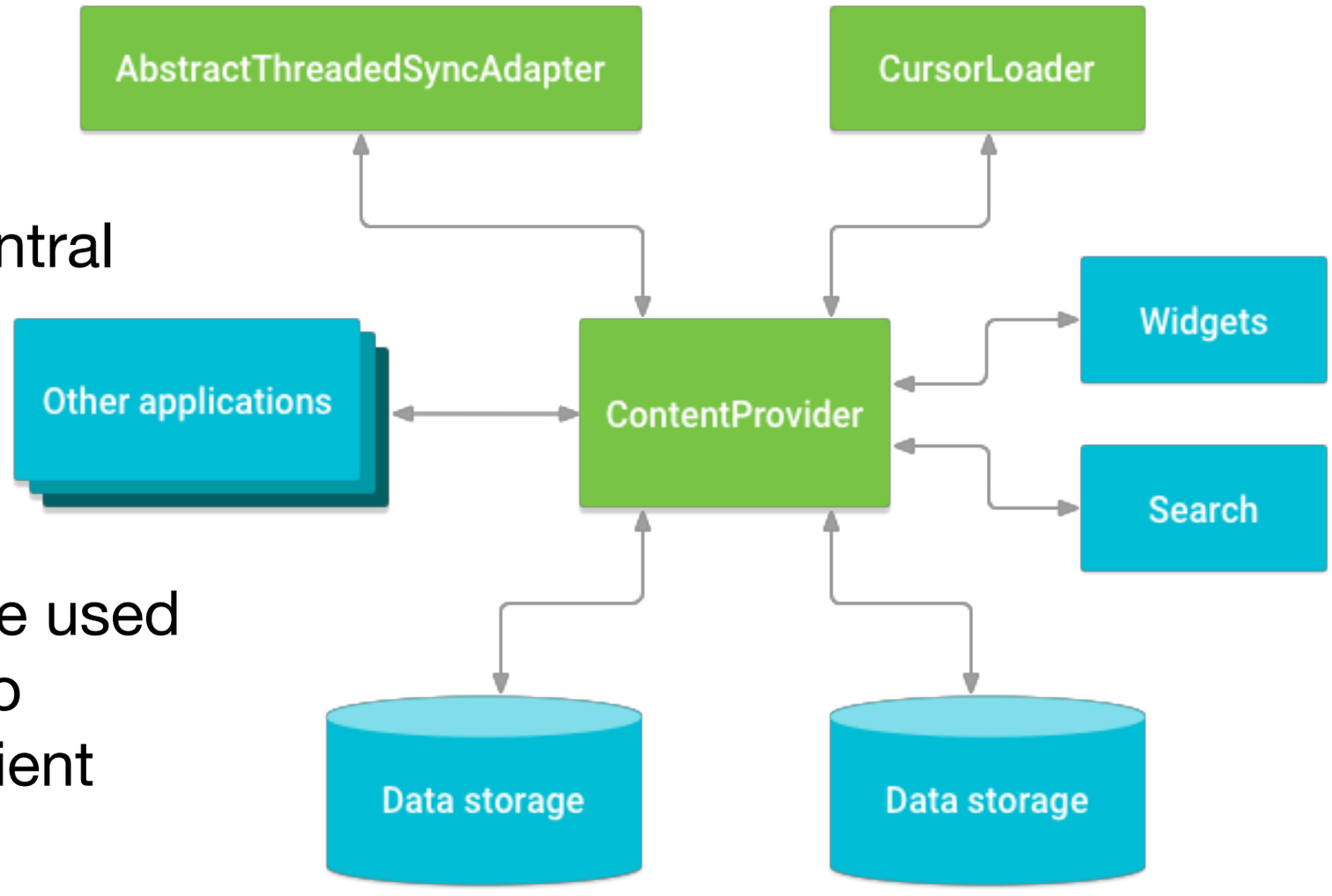
- LiveData
- ModelView
- Content Providers
- Loaders
- Sync Adapters
- WebSockets
- Notifications

Synchronizing Data

- LiveData
- ModelView
- Content Providers
- Loaders
- Sync Adapters
- WebSockets
- Notifications

Content Provider

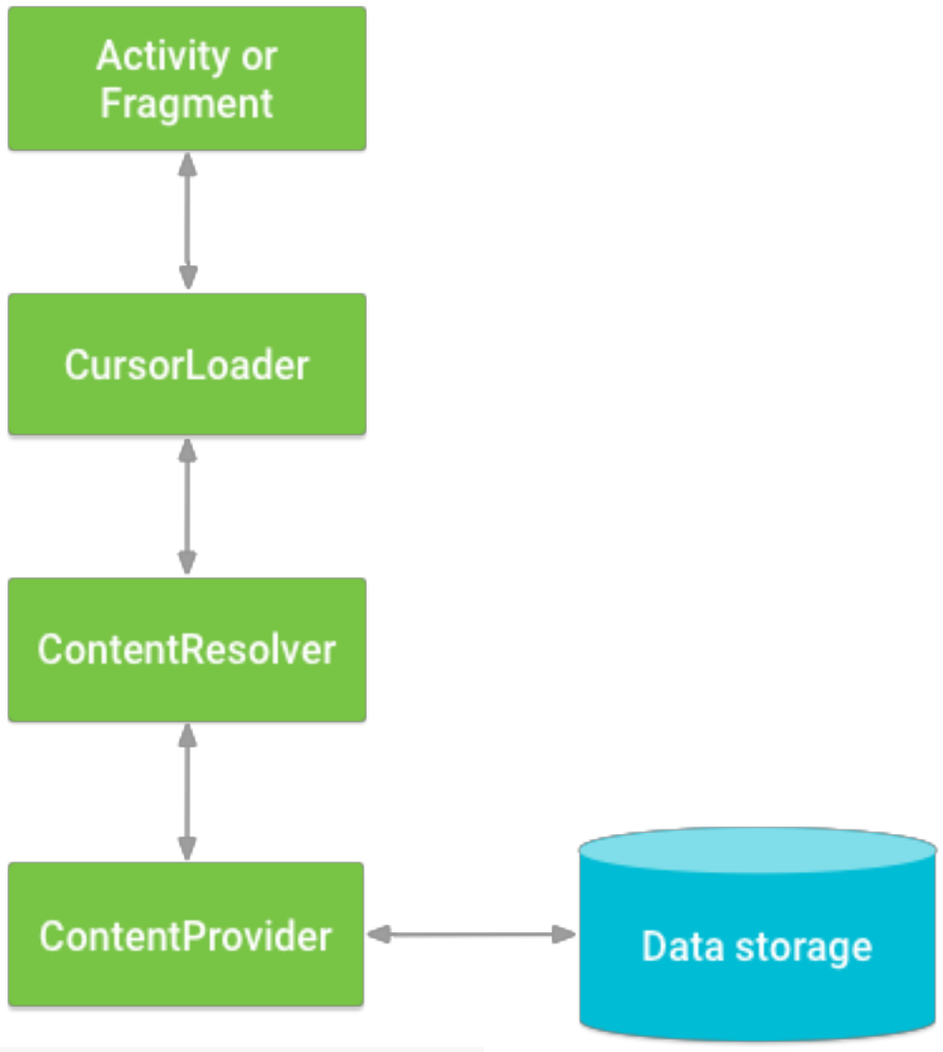
- Manages access to central repository of data.
- Part of the application.
- Primarily intended to be used by other applications to access the provided client objects.



https://developer.android.com/guide/topics/providers/content-provider-basics

Accessing a provider

word	app id	frequency	locale	_ID
mapreduce	user1	100	en_US	1
precompiler	user14	200	fr_FR	2
applet	user2	225	fr_CA	3
const	user1	255	pt_BR	4
int	user5	100	en_UK	5




```
// Queries the user dictionary and returns results
mCursor = contentResolver.query(
        UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
        mProjection,                        // The columns to return for each row
        mSelectionClause,                   // Selection criteria
        mSelectionArgs.toTypedArray(),      // Selection criteria
        mSortOrder                          // The sort order for the returned rows
)
```

Constructing the query

Constructing the query

- Requesting read access permission

```
<uses-permission android:name="android.permission.READ_USER_DICTIONARY">
```

Constructing the query

- Requesting read access permission

```
<uses-permission android:name="android.permission.READ_USER_DICTIONARY">
```

- Constructing the query

```
// A "projection" defines the columns that will be returned for each row
private val mProjection: Array<String> = arrayOf(
 UserDictionary.Words._ID,    // Contract constant for the _ID
 UserDictionary.Words.WORD,   // Contract constant for the word
 UserDictionary.Words.LOCALE  // Contract constant for the locale
)

// Defines a string to contain the selection clause
private var mSelectionClause: String? = null

// Declares an array to contain selection arguments
private lateinit var mSelectionArgs: Array<String>
```

Constructing the query

```
/*
 * This declares String array to contain the selection arguments.
 */
private lateinit var mSelectionArgs: Array<String>

// Gets a word from the UI
mSearchString = mSearchWord.text.toString()

// Remember to insert code here to check for invalid or malicious input.
```

```
// If the word is the empty string, gets everything
mSelectionArgs = mSearchString?.takeIf { it.isNotEmpty() }?.let {
 mSelectionClause = "${UserDictionary.Words.WORD} = ?"
 arrayOf(it)
} ?: run {
 mSelectionClause = null
 emptyArray<String>()
}

// Does a query against the table and returns a Cursor object
mCursor = contentResolver.query(
 UserDictionary.Words.CONTENT_URI,  // The content URI of the words table
 mProjection,                       // The columns to return for each row
 mSelectionClause,                  // Either null, or the word the user entered
 mSelectionArgs,                    // Either empty, or the string the user entered
 mSortOrder,                        // The sort order for the returned rows
```

```
// Does a query against the table and returns a Cursor object
mCursor = contentResolver.query(
 UserDictionary.Words.CONTENT_URI,  // The content URI of the words table
 mProjection,                       // The columns to return for each row
 mSelectionClause,                  // Either null, or the word the user entered
 mSelectionArgs,                    // Either empty, or the string the user entered
 mSortOrder                         // The sort order for the returned rows
)

// Some providers return null if an error occurs, others throw an exception
when (mCursor?.count) {
 null -> {
 /*
  * Insert code here to handle the error. Be sure not to use the cursor!
  * You may want to call android.util.Log.e() to log this error.
  */
 }
 0 -> {
 /*
  * Insert code here to notify the user that the search was unsuccessful. This
isn't
  * necessarily an error. You may want to offer the user the option to insert a new
  * row, or re-type the search term.
  */
 }
 else -> {
  // Insert code here to do something with the results
 }
}

SELECT _ID, word, locale FROM words WHERE word = <userinput> ORDER BY word ASC;
```

Inserting, updating, and deleting data

Inserting, updating, and deleting data

- Insert

```
// Defines a new Uri object that receives
// the result of the insertion
lateinit var mNewUri: Uri
// ...
// Defines an object to contain the new values to insert
val mNewValues = ContentValues().apply {
/*
 * Sets the values of each column and inserts the word.
 * The arguments to the "put" method are "column name"
 * and "value"
 */
 put(UserDictionary.Words.APP_ID, "example.user")
 put(UserDictionary.Words.LOCALE, "en_US")
 put(UserDictionary.Words.WORD, "insert")
 put(UserDictionary.Words.FREQUENCY, "100")
}

mNewUri = contentResolver.insert(
 // the user dictionary content URI
 UserDictionary.Words.CONTENT_URI,
 // the values to insert
 mNewValues
)
```

Inserting, updating, and deleting data

- Insert

```
// Defines a new Uri object that receives
// the result of the insertion
lateinit var mNewUri: Uri
// ...
// Defines an object to contain the new values to insert
val mNewValues = ContentValues().apply {
/*
 * Sets the values of each column and inserts the word.
 * The arguments to the "put" method are "column name"
 * and "value"
 */
 put(UserDictionary.Words.APP_ID, "example.user")
 put(UserDictionary.Words.LOCALE, "en_US")
 put(UserDictionary.Words.WORD, "insert")
 put(UserDictionary.Words.FREQUENCY, "100")
}
mNewUri = contentResolver.insert(
 // the user dictionary content URI
 UserDictionary.Words.CONTENT_URI,
 // the values to insert
 mNewValues
)
```

content://user_dictionary/words/<id_value>

Inserting, updating, and deleting data

- Insert
- Update

```
// Defines an object to contain the updated values
val mUpdateValues = ContentValues().apply {
 // Sets the updated value and updates the selected words.
 putNull(UserDictionary.Words.LOCALE)
}
// Defines selection criteria for the rows
// you want to update
val mSelectionClause: String =
  UserDictionary.Words.LOCALE + "LIKE ?"
val mSelectionArgs: Array<String> = arrayOf("en_%")

// Defines a variable to contain the
// number of updated rows
var mRowsUpdated: Int = 0
//...
mRowsUpdated = contentResolver.update(
UserDictionary.Words.CONTENT_URI,
 mUpdateValues,
 mSelectionClause,
 mSelectionArgs
)
```

Inserting, updating, and deleting data

- Insert
- Update
- Delete

```
// Defines selection criteria for the rows you want to delete
val mSelectionClause = "${UserDictionary.Words.LOCALE} LIKE ?"
val mSelectionArgs: Array<String> = arrayOf("user")
```

```
// Defines a variable to contain the number of rows deleted
var mRowsDeleted: Int = 0

// ...

// Deletes the words that match the selection criteria
mRowsDeleted = contentResolver.delete(
        UserDictionary.Words.CONTENT_URI,
        mSelectionClause,
        mSelectionArgs
)
```

DEMO

Loaders

- Deprecated as of Pie (API 28)
- Load data from a content provider or other sources.
- Problems that are solving:
  - Fetches data directly in the activity or fragment.
    - Need to manage the thread lifecycle.
- Run on separate threads to prevent janky or unresponsive UI.
- Simplify thread management by providing callback methods when events occur.
- Persist and cache results across configuration.

https://developer.android.com/guide/components/loaders

DEMO

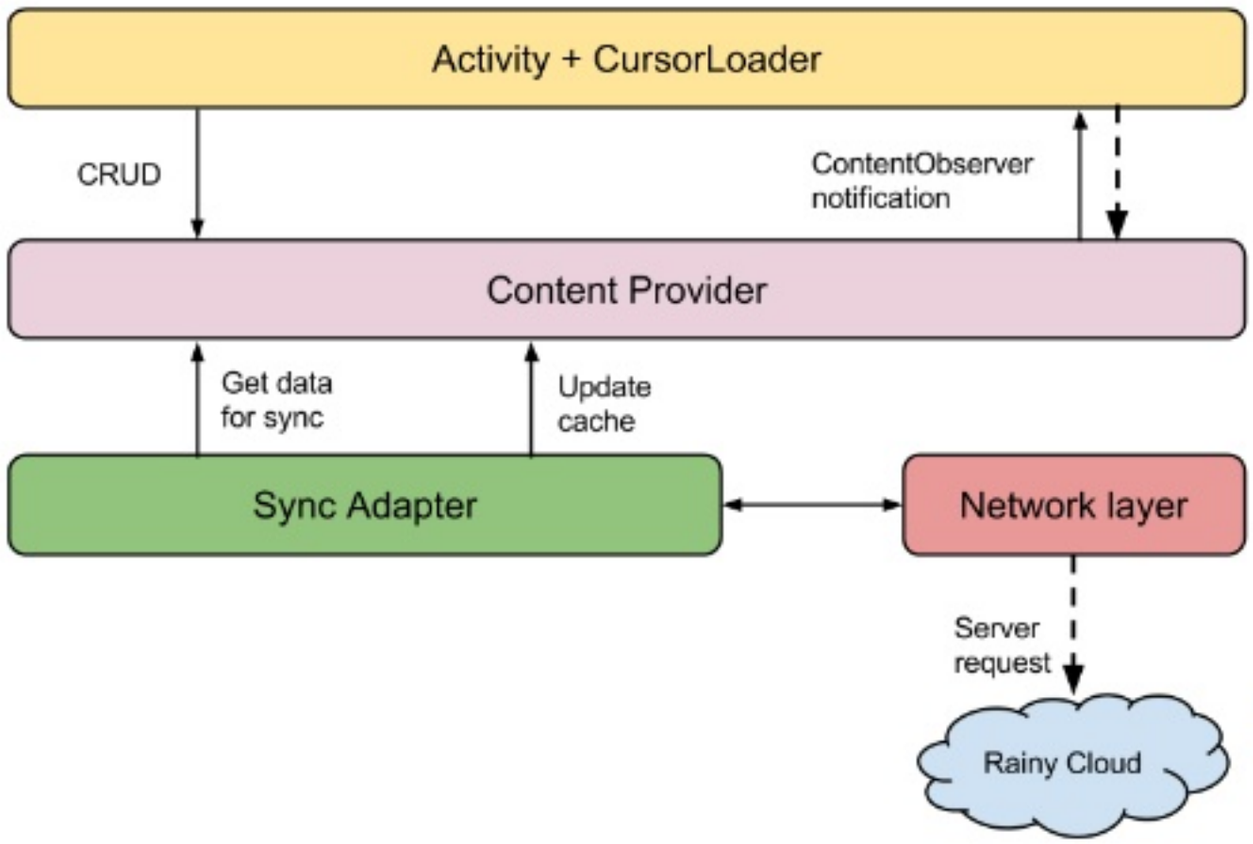
Loaders

- Deprecated as of Pie (API 28)
- Load data from a content provider or other sources.
- Problems that are solving:
  - Fetches data directly in the activity or fragment
  - Need to manage the thread lifecycle
- Run on separate threads to prevent janky or unresponsive UI.
- Simplify thread management by providing callback methods when events occur.
- Persist and cache results across configuration.

https://developer.android.com/guide/components/loaders

Transfer data using sync adapters

- Plug-in architecture.
- Automated execution.
- Automated network checking.
- Improved battery performance.
- Account management and authentication.



https://developer.android.com/training/sync-adapters/

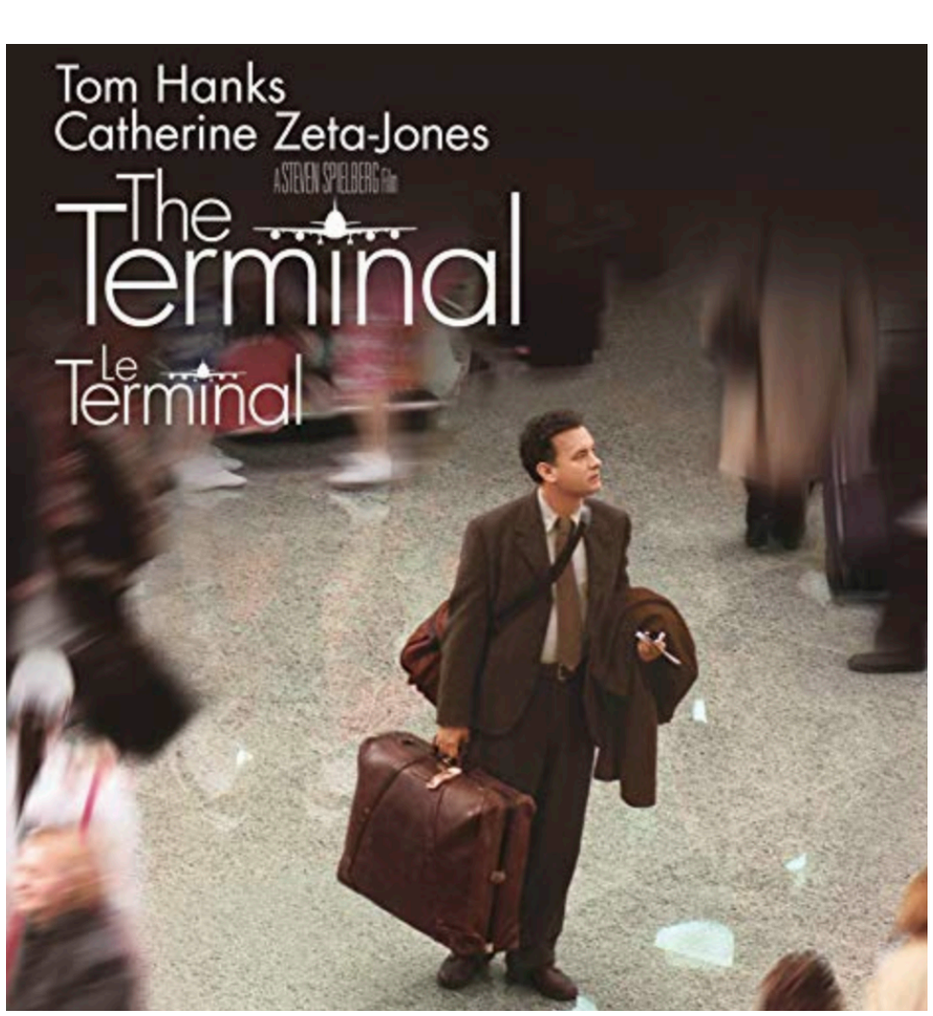
Sync adapters run asynchronously, so you should use them with the expectation that they transfer data regularly and efficiently, but not instantaneously. If you need to do real-time data transfer, you should do it in an AsyncTask or an IntentService.

Create the sync adapter

```
/**
 * Handle the transfer of data between a server and an
 * app, using the Android sync adapter framework.
 */
class SyncAdapter @JvmOverloads constructor(
 context: Context,
 autoInitialize: Boolean,
 /**
 * Using a default argument along with @JvmOverloads
 * generates constructor for both method signatures to maintain compatibility
 * with Android 3.0 and later platform versions
 */
 allowParallelSyncs: Boolean = false,
 /*
 * If your app uses a content resolver, get an instance of it
 * from the incoming Context
 */
 val mContentResolver: ContentResolver = context.contentResolver
) : AbstractThreadedSyncAdapter(context, autoInitialize, allowParallelSyncs) {
    // ...
}
```

Add the data transfer code

```
/*
 * Specify the code you want to run in the sync adapter. The entire
 * sync adapter runs in a background thread, so you don't have to set
 * up your own background processing.
 */
override fun onPerformSync(
 account: Account,
 extras: Bundle,
 authority: String,
 provider: ContentProviderClient,
 syncResult: SyncResult
) {
 /*
 * Put the data transfer code here.
 */
}
```



Bind the sync adapter to the framework

```
/**
 * Define a Service that returns an [android.os.IBinder] for the
 * sync adapter class, allowing the sync adapter framework to call
 * onPerformSync().
 */
class SyncService : Service() {
 /*
  * Instantiate the sync adapter object.
  */
 override fun onCreate() {
  /*
   * Create the sync adapter as a singleton.
   * Set the sync adapter as syncable
   * Disallow parallel syncs
   */
  synchronized(sSyncAdapterLock) {
   sSyncAdapter = sSyncAdapter ?: SyncAdapter(applicationContext, true)
  }
 }
```

```
    * Set the sync adapter as syncable
     * Disallow parallel syncs
     */
    synchronized(sSyncAdapterLock) {
     sSyncAdapter = sSyncAdapter ?: SyncAdapter(applicationContext, true)
    }
  }

  /**
   * Return an object that allows the system to invoke
   * the sync adapter.
   */
  override fun onBind(intent: Intent): IBinder {
   /*
    * Get the object that allows external processes
    * to call onPerformSync(). The object is created
    * in the base class code when the SyncAdapter
    * constructors call super()
    *
    * We should never be in a position where this is called before
    * onCreate() so the exception should never be thrown
    */
    return sSyncAdapter?.syncAdapterBinder ?: throw IllegalStateException()
   }

   companion object {
    // Storage for an instance of the sync adapter
    private var sSyncAdapter: SyncAdapter? = null
    // Object to use as a thread-safe lock
    private val sSyncAdapterLock = Any()
   }
  }
```

Add the account required by the framework

```
// Constants
// The authority for the sync adapter's content provider
const val AUTHORITY = "com.example.android.datasync.provider"
// An account type, in the form of a domain name
const val ACCOUNT_TYPE = "example.com"
// The account name
const val ACCOUNT = "dummyaccount"
```

```
class MainActivity : FragmentActivity() {
 // Instance fields
 private lateinit var mAccount: Account
 ...
 override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  ...
  // Create the dummy account
  mAccount = createSyncAccount()
  ...
 }
 ...
 /**
  * Create a new dummy account for the sync adapter
  */
```

```
 // Create the dummy account
 mAccount = createSyncAccount()
 ...
}
...
/**
 * Create a new dummy account for the sync adapter
 */
private fun createSyncAccount(): Account {
 val accountManager = getSystemService(Context.ACCOUNT_SERVICE) as AccountManager
 return Account(ACCOUNT, ACCOUNT_TYPE).also { newAccount ->
  /*
   * Add the account and account type, no password or user data
   * If successful, return the Account object, otherwise report an error.
   */
  if (accountManager.addAccountExplicitly(newAccount, null, null)) {
   /*
    * If you don't set android:syncable="true" in
    * in your <provider> element in the manifest,
    * then call context.setIsSyncable(account, AUTHORITY, 1)
    * here.
    */
  } else {
   /*
    * The account exists or some other error occurred.
    * Log this, report it, or handle it internally.
    */
  }
 }
}
...
}
```

Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:contentAuthority="com.example.android.datasync.provider"
 android:accountType="com.android.example.datasync"
 android:userVisible="false"
 android:supportsUploading="false"
 android:allowParallelSyncs="false"
 android:isAlwaysSyncable="true"/>
```

Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:contentAuthority="com.example.android.datasync.provider"
android:accountType="com.android.example.datasync"
android:userVisible="false"
android:supportsUploading="false"
android:allowParallelSyncs="false"
android:isAlwaysSyncable="true"/>
```

Sets the visibility of the sync adapter's account type in the **Accounts** section of the system's Settings app.

Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:contentAuthority="com.example.android.datasync.provider"
android:accountType="com.android.example.datasync"
android:userVisible="false"
android:supportsUploading="false"
android:allowParallelSyncs="false"
android:isAlwaysSyncable="true"/>
```

Allows you to upload data to the cloud.
Set this to `false` if your app only downloads data.

Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:contentAuthority="com.example.android.datasync.provider"
 android:accountType="com.android.example.datasync"
 android:userVisible="false"
 android:supportsUploading="false"
 android:allowParallelSyncs="false"
 android:isAlwaysSyncable="true"/>
```

Allows multiple instances of your sync adapter component to run at the same time. Use this if your app supports multiple user accounts.

Manifest

```xml
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:contentAuthority="com.example.android.datasync.provider"
 android:accountType="com.android.example.datasync"
 android:userVisible="false"
 android:supportsUploading="false"
 android:allowParallelSyncs="false"
 android:isAlwaysSyncable="true"/>
```

Run your sync adapter at any time you've specified.
If you want to programmatically control when your sync adapter can run,
set this flag to `false`, and then call`requestSync()` to run the sync adapter.

https://developer.android.com/training/sync-adapters/running-sync-adapter

Run a sync adapter

Run a sync adapter

- When server data changes.
- When device data changes.
- At regular intervals.
- On demand.

When server data changes

```
// Content provider authority
const val AUTHORITY = "com.example.android.datasync.provider"
// Account type
const val ACCOUNT_TYPE = "com.example.android.datasync"
// Account
const val ACCOUNT = "default_account"
// Incoming Intent key for extended data
const val KEY_SYNC_REQUEST = "com.example.android.datasync.KEY_SYNC_REQUEST"
```

```
...
class GcmBroadcastReceiver : BroadcastReceiver() {
 ...
 override fun onReceive(context: Context, intent: Intent) {
  // Get a GCM object instance
  val gcm: GoogleCloudMessaging = GoogleCloudMessaging.getInstance(context)
  // Get the type of GCM message
  val messageType: String? = gcm.getMessageType(intent)
  /*
   * Test the message type and examine the message contents.
   * Since GCM is a general-purpose messaging system, you
   * may receive normal messages that don't require a sync
   * adapter run.
   * The following code tests for a a boolean flag indicating
   * that the message is requesting a transfer from the device.
   */
```

```
const val ACCOUNT = "default_account"
// Incoming Intent key for extended data
const val KEY_SYNC_REQUEST = "com.example.android.datasync.KEY_SYNC_REQUEST"
...
class GcmBroadcastReceiver : BroadcastReceiver() {
 ...
 override fun onReceive(context: Context, intent: Intent) {
  // Get a GCM object instance
  val gcm: GoogleCloudMessaging = GoogleCloudMessaging.getInstance(context)
  // Get the type of GCM message
  val messageType: String? = gcm.getMessageType(intent)
  /*
   * Test the message type and examine the message contents.
   * Since GCM is a general-purpose messaging system, you
   * may receive normal messages that don't require a sync
   * adapter run.
   * The following code tests for a a boolean flag indicating
   * that the message is requesting a transfer from the device.
   */
  if (GoogleCloudMessaging.MESSAGE_TYPE_MESSAGE == messageType
   && intent.getBooleanExtra(KEY_SYNC_REQUEST, false)) {
   /*
    * Signal the framework to run your sync adapter. Assume that
    * app initialization has already created the account.
    */
   ContentResolver.requestSync(mAccount, AUTHORITY, null)
   ...
  }
  ...
 }
 ...
}
```

Run the sync adapter when content provider data changes

```
/*
 * Define a method that's called when data in the
 * observed content provider changes.
 * This method signature is provided for compatibility with
 * older platforms.
 */
override fun onChange(selfChange: Boolean) {
 /*
  * Invoke the method signature available as of
  * Android platform version 4.1, with a null URI.
  */
  onChange(selfChange, null)
}
/*
 * Define a method that's called when data in the
 * observed content provider changes.
 */
override fun onChange(selfChange: Boolean, changeUri: Uri?) {
 /*
  * Ask the framework to run your sync adapter.
  * To maintain backward compatibility, assume that
  * changeUri is null.
  */
 ContentResolver.requestSync(mAccount, AUTHORITY, null)
}
```

Run the sync adapter periodically

```
// Content provider authority
const val AUTHORITY = "com.example.android.datasync.provider"
// Account
const val ACCOUNT = "default_account"
// Sync interval constants
const val SECONDS_PER_MINUTE = 60L
const val SYNC_INTERVAL_IN_MINUTES = 60L
const val SYNC_INTERVAL = SYNC_INTERVAL_IN_MINUTES * SECONDS_PER_MINUTE
...
class MainActivity : FragmentActivity() {
 ...
 // A content resolver for accessing the provider
 private lateinit var mResolver: ContentResolver

 override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  ...
  // Get the content resolver for your app
  mResolver = contentResolver
  /*
   * Turn on periodic syncing
   */
   ContentResolver.addPeriodicSync(
     mAccount,
```

```
// Content provider authority
const val AUTHORITY = "com.example.android.datasync.provider"
// Account
const val ACCOUNT = "default_account"
// Sync interval constants
const val SECONDS_PER_MINUTE = 60L
const val SYNC_INTERVAL_IN_MINUTES = 60L
const val SYNC_INTERVAL = SYNC_INTERVAL_IN_MINUTES * SECONDS_PER_MINUTE
...
class MainActivity : FragmentActivity() {
 ...
 // A content resolver for accessing the provider
 private lateinit var mResolver: ContentResolver

 override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  ...
  // Get the content resolver for your app
  mResolver = contentResolver
  /*
   * Turn on periodic syncing
   */
   ContentResolver.addPeriodicSync(
     mAccount,
     AUTHORITY,
     Bundle.EMPTY,
     SYNC_INTERVAL)
    ...
 }
 ...
}
```

Run the sync adapter on demand

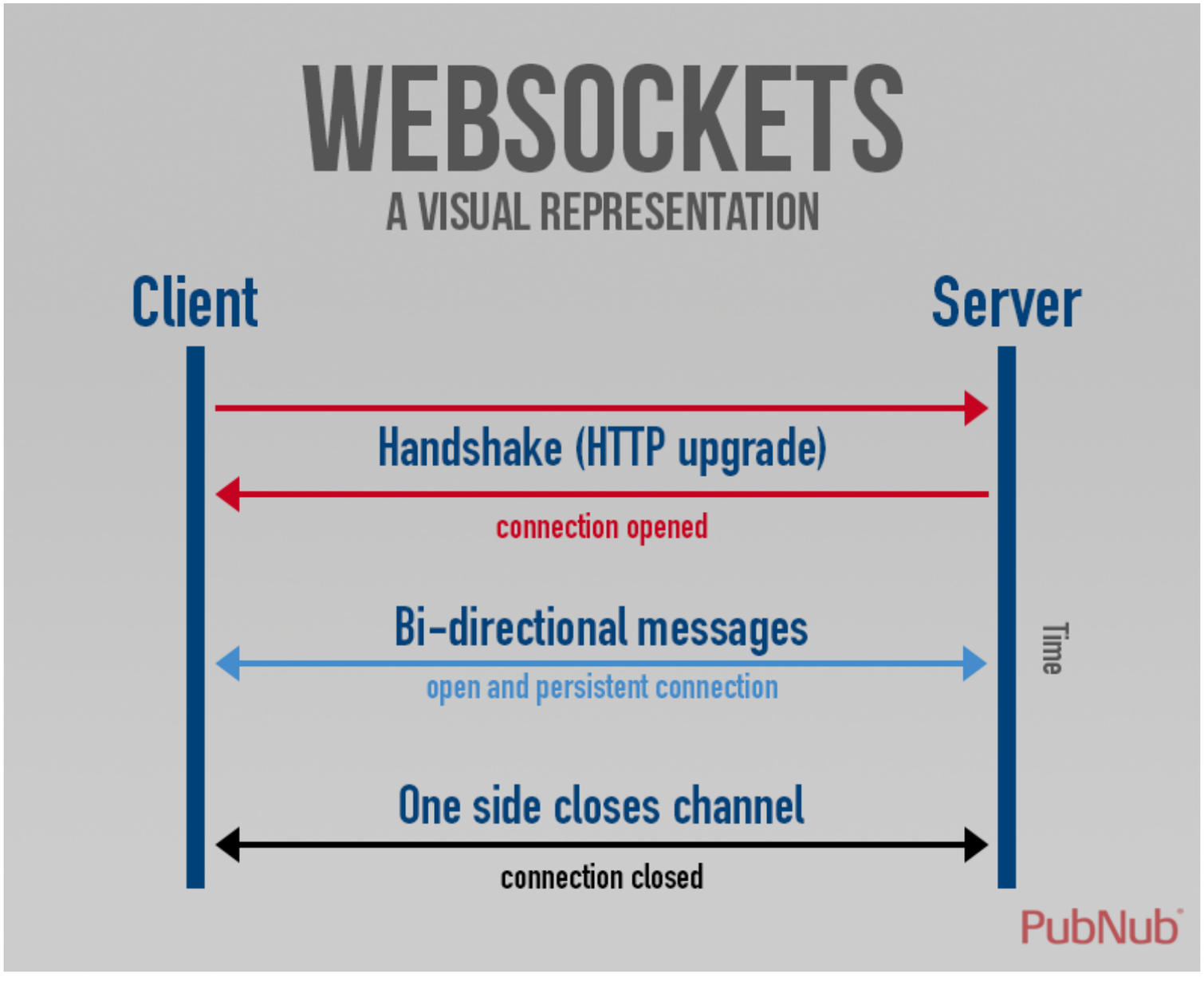

```
// Constants
// Content provider authority
val AUTHORITY = "com.example.android.datasync.provider"
// Account type
val ACCOUNT_TYPE = "com.example.android.datasync"
// Account
val ACCOUNT = "default_account"
```

```
...
class MainActivity : FragmentActivity() {
 ...
 // Instance fields
 private lateinit var mAccount: Account
 ...
 override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  ...
  /*
   * Create the dummy account. The code for CreateSyncAccount
   * is listed in the lesson Creating a Sync Adapter
   */
  mAccount = createSyncAccount()
  ...
 }
```

DEMO

```
        super.onCreate(savedInstanceState)
        ...
        /*
         * Create the dummy account. The code for CreateSyncAccount
         * is listed in the lesson Creating a Sync Adapter
         */
        mAccount = createSyncAccount()
        ...
    }

    /**
     * Respond to a button click by calling requestSync(). This is an
     * asynchronous operation.
     *
     * This method is attached to the refresh button in the layout
     * XML file
     *
     * @param v The View associated with the method call,
     * in this case a Button
     */
    fun onRefreshButtonClick(v: View) {
        // Pass the settings flags by inserting them in a bundle
        val settingsBundle = Bundle().apply {
            putBoolean(ContentResolver.SYNC_EXTRAS_MANUAL, true)
            putBoolean(ContentResolver.SYNC_EXTRAS_EXPEDITED, true)
        }
        /*
         * Request the sync for the default account, authority, and
         * manual sync settings
         */
        ContentResolver.requestSync(mAccount, AUTHORITY, settingsBundle)
    }
```

WebSocket


WEBSOCKETS
A VISUAL REPRESENTATION
Client
Server
Handshake (HTTP upgrade)
connection opened
Bi-directional messages
open and persistent connection
One side closes channel
connection closed
Time
PubNub

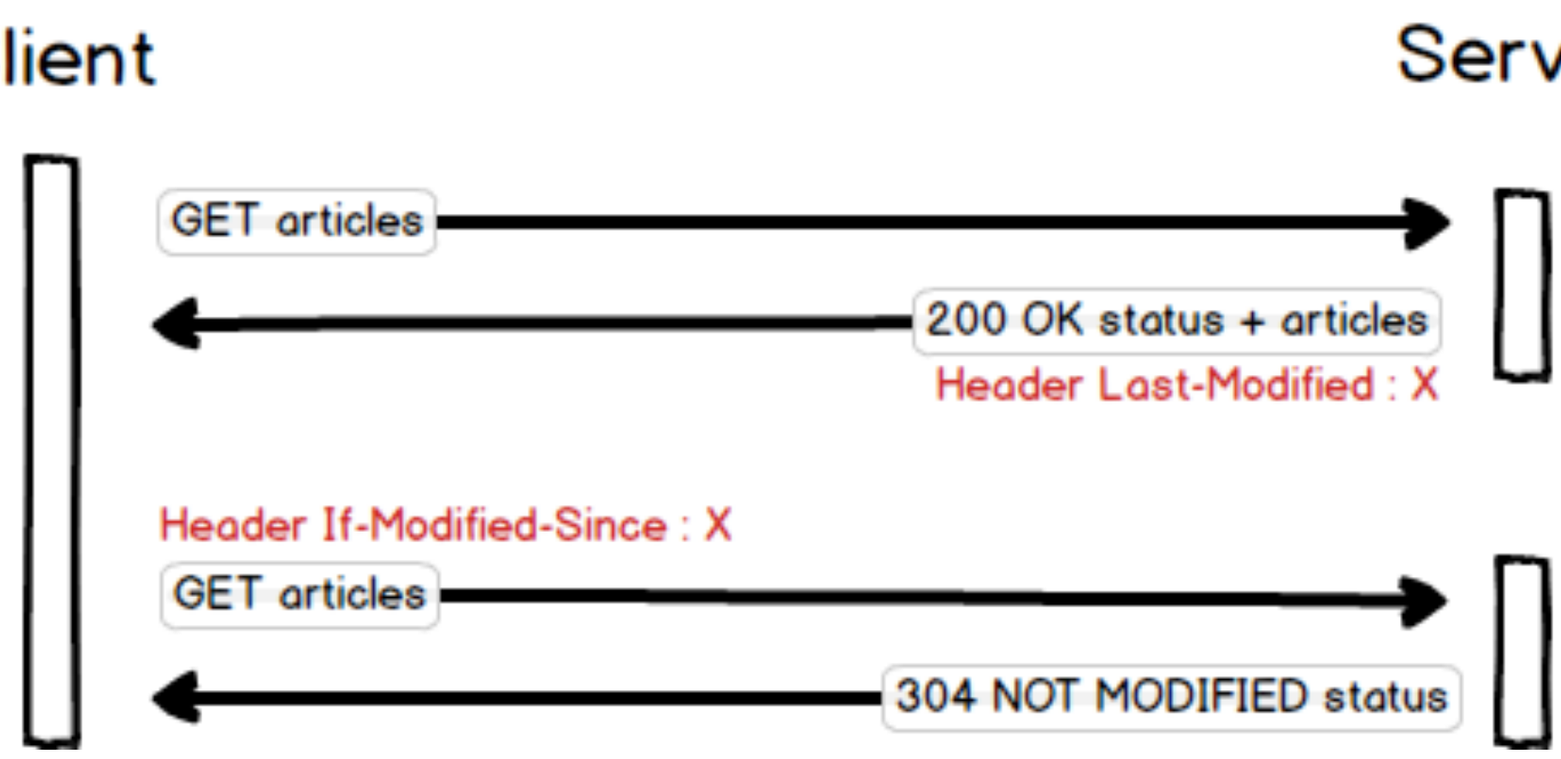
Okhttp

```
val client = OkHttpClient.Builder()
  .readTimeout(3, TimeUnit.SECONDS)
  .build()
val request = Request.Builder()
  .url("ws://10.0.2.2:3000")
  .build()
val wsListener = EchoWebSocketListener()
client.newWebSocket(request, wsListener)
```


```
private class EchoWebSocketListener : WebSocketListener() {
 override fun onOpen(webSocket: WebSocket, response: Response) {
  webSocket.send("Hello, there!")
  webSocket.close(NORMAL_CLOSURE_STATUS, "Goodbye !")
 }

 override fun onMessage(webSocket: WebSocket?, text: String?) {
  logd("Receiving : ${text!!}")
 }

 override fun onMessage(webSocket: WebSocket?, bytes: ByteString?) {
  logd("Receiving bytes : ${bytes!!.hex()}")
 }

 override fun onClosing(webSocket: WebSocket?, code: Int, reason: String?) {
  webSocket!!.close(NORMAL_CLOSURE_STATUS, null)
  logd("Closing : $code / $reason")
 }

 override fun onFailure(webSocket: WebSocket, t: Throwable, response: Response?) {
  logd("Error : ${t.message}", t)
 }

 companion object {
  private const val NORMAL_CLOSURE_STATUS = 1000
 }
}
```

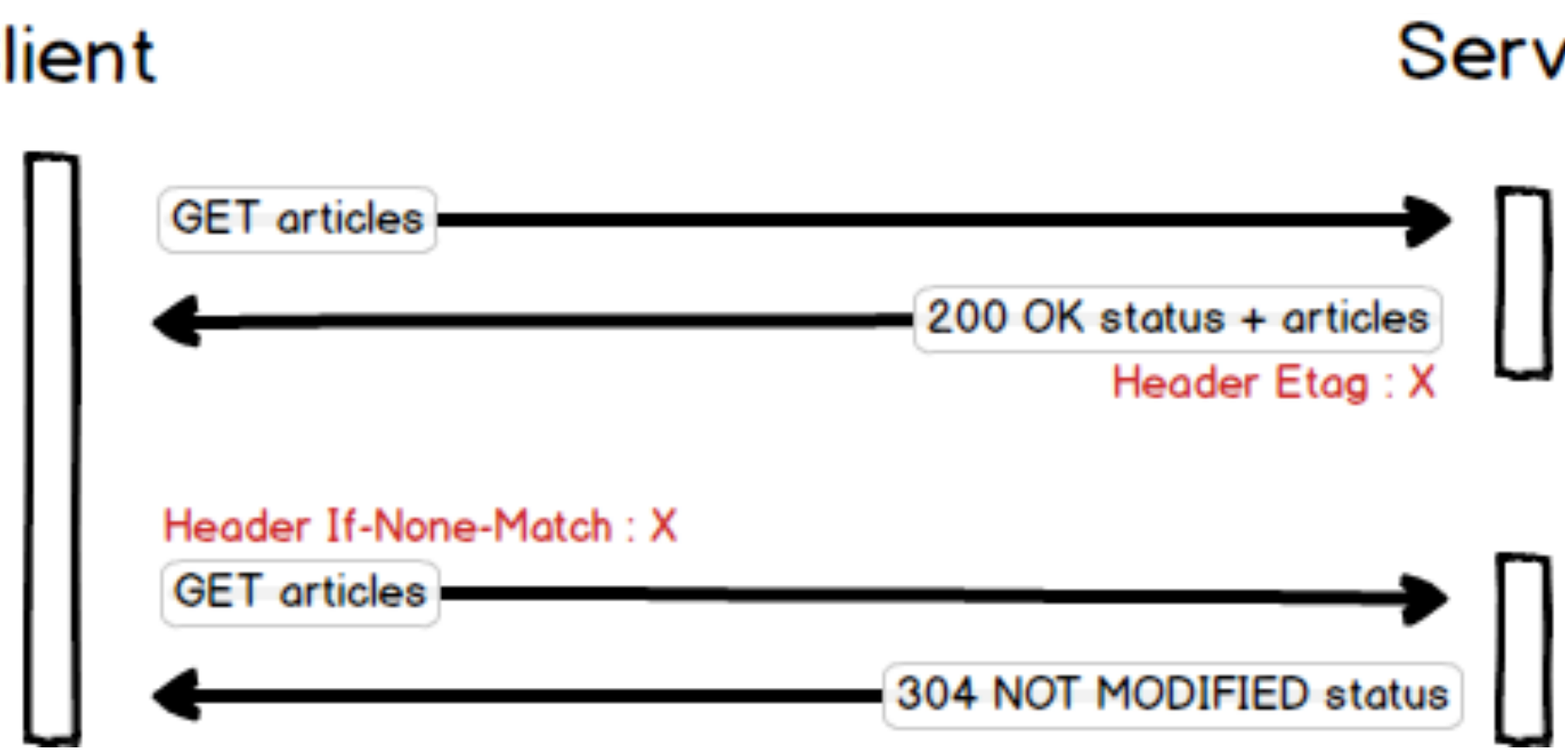
https://square.github.io/okhttp/

OkHttp, Etags and If-Modified-Since



http://square.github.io/retrofit/

OkHttp, Etags and If-Modified-Since



http://square.github.io/retrofit/

Okhttp Cache

```
private final static int CACHE_SIZE_BYTES = 1024 * 1024 * 2;
public static Retrofit getAdapter(Context context, String baseUrl) {
    OkHttpClient.Builder builder = new OkHttpClient().newBuilder();
    builder.cache(
        new Cache(context.getCacheDir(), CACHE_SIZE_BYTES));
    OkHttpClient client = builder.build();
    Retrofit.Builder retrofitBuilder = new Retrofit.Builder();
    retrofitBuilder.baseUrl(baseUrl).client(client);
    return retrofitBuilder.build();
}
```

The ***Last-Modified*** headers or ***Etags*** will be automatically used depending on the servers responses.

Reduce processing

```
if (response.isSuccessful() &&
    response.raw().networkResponse() != null &&
    response.raw().networkResponse().code() ==
            HttpURLConnection.HTTP_NOT_MODIFIED) {
    // not modified, no need to do anything.
    return;
}
// parse response here
```

Lecture outcomes

- Use the following, to retrieve and synchronize data:
  - Content Providers.
  - Loaders.
  - Sync Adapters.
- Use HTTP ETag to save bandwidth.
- Use web sockets to notify the clients about server updates.