ForecastRestYahoo
A REST Architecture
Première version

Ce projet fait partie d'un ensemble de projets dont l'objectif est d'étudier les différentes architectures possibles pour une application Android.

L'objectif de ce document est de vous permettre de comprendre

l'architecture du projet ForecastYahooRest première version.

Auteurs:

Mathias Seguy

Date de création 11/10/2014

Version 1.0

1 Principes

Ce projet a pour objectif d'étudier et d'expliquer les différentes architectures pouvant être mises en place pour effectuer un appel Rest dans le but d'afficher les données renvoyées. Le projet ForecastYahooRest est la première architecture possible, d'autres suivront. On verra quels sont les problèmes et les avantages de ces architectures et nous les ferons évoluer en fonction de ceux-ci.

Même si cette architecture est la première que présentée, elle est propre et respecte les bons principes d'architecture à mettre en place sur Android pour les appels Rest. Elle s'appuie sur la conférence donnée par Mathias Seguy au PAUG, TAUG, "An Android Journey", les slides sont disponibles sur SliderShare (ici) et la conférence peut se voir sur YouTube, sur la chaîne du ParisAndroidUserGroup (ici) ou d'Android2EE (ici).

L'objectif de ces architectures est de mettre en place le principe suivant:

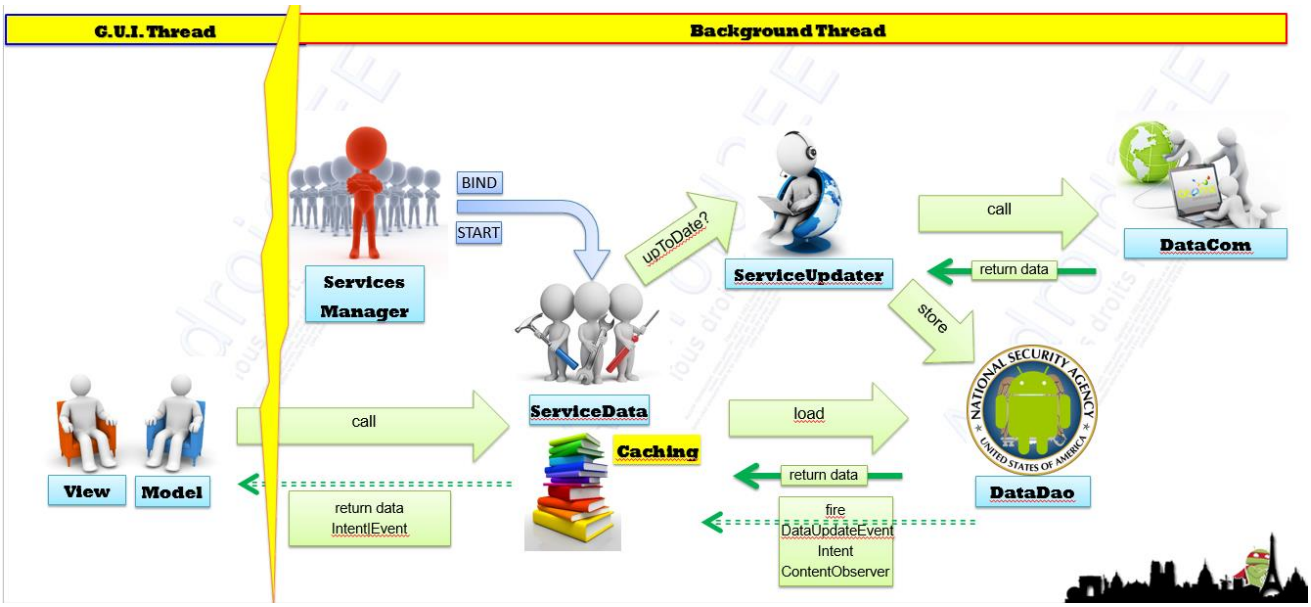

Figure 1: Principe d'appel réseau

Dans cette première version, la mise en place des éléments suivants a été effectuée:

- Utilisation d'un BrodacastReceiver pour être au courant en permanence de l'état de la connectivité réseau. Persistance de cette information par l'objet Application.
- Mise en place de listener pour écouter les changements de connectivité réseau au sein des différentes classes de l'application.
- Centralisation de la gestion des services avec le ServiceManager (accès, destruction).
- Persistance et accès au ServiceManager par l'objet Application et mise en place du 1Second µPattern pour la destruction des services.
- Mise en place de la centralisation de la gestion des Exceptions via un ExceptionManager et des ExceptionManaged. Affichage automatique des Exceptions via la MotherActivity.
- Mise en place d'une base de données pour le caching des données et de sa D.A.O. associée.
- Mise en place de deux services, l'un le ServiceData gère les données, le second le ServiceUpdater les met à jour.

- L'utilisation de CallBack a été implémentée pour la récupération des données, plutôt que celle de la communication par Intent.

Ci-dessous, le principe de mise en place du ServiceManager avec l'utilisation de CallBack.

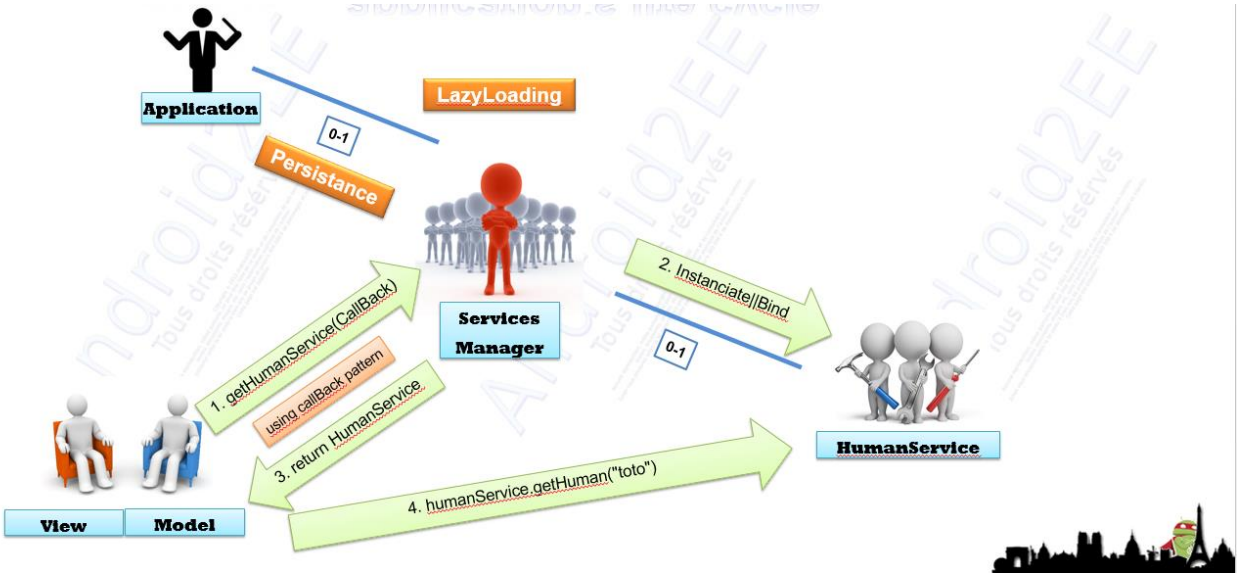

Figure 2: Principe du ServiceManager

Nous allons, dans la suite de ce document, voir la mise en place de ces différents principes architecturaux au sein de ce projet.

2 L'état de la connexion réseau.

Un principe fort, au sein de vos applications, doit être la connaissance de l'état de la connectivité réseau. En effet, s'il n'y pas de connexion, est-il utile de faire un appel REST ? pas vraiment, hein. Il est donc nécessaire de connaître cet état en permanence où que vous vous trouviez dans l'application. Nous verrons dans les projets suivants comment mettre en place le pattern d'état associé à cette problématique, mais pour l'instant, nous mettons en place un système plus simpliste de cette gestion.

L'idée générale est de déclarer un BroadcastReceiver qui écoute les changements d'état du réseau et utilise l'objet Application pour centraliser cet état. L'objet Application a, lui, à charge de ventiler l'information aux autres éléments de votre application.

2.1 Le BroadCastReceiever : ConnectionReceiver

Ce BroadcastReceiver doit écouter tous les changements de connectivité réseau et mettre à jour l'objet Application.

```
<!-- BroadCastReceiver -->
    <!-- Listening for Boot and connectivity changed (Launch the update service) -->
    <receiver android:name=".broadcastreceiver.WebConnectivityReceiver" >
      <intent-filter>
        <action android:name="android.intent.action.ACTION_DOCK_EVENT" />
        <action android:name="android.intent.action.ACTION_BATTERY_LOW" />
```

```
            <action android:name="android.intent.action.ACTION_POWER_CONNECTED" />
            <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED" />
            <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
            <action android:name="android.intent.action.BOOT_COMPLETED" />
            <action android:name="android.net.wifi.STATE_CHANGE" />
        </intent-filter>
    </receiver>
```

Ainsi le système instancie ce receiver lorsque l'un de ces évènements système est déclenché. Il doit alors prévenir l'objet Activity:

```
public class WebConnectivityReceiver extends BroadcastReceiver {
        /*
         * (non-Javadoc)
         * @see android.content.BroadcastReceiver#onReceive(android.content.Context,
         * android.content.Intent)
         */
        @Override
        public void onReceive(Context context, Intent intent) {
                Log.i("WebConnectivityReceiver", "Intent received :" + intent.getAction());
                MyApplication.instance.manageConnectivtyState();

        }
}
```

2.2 L'objet Application

L'objet application est alors appelé par le receiver et doit, analyser l'état de cette connectivité, la persister (pour pouvoir s'en souvenir quand l'application sera en cours d'exécution) et ventiler l'information (si l'application est en cours d'exécution).

2.2.1 Analyse et persistance de l'état réseau

L'analyse de l'état réseau est assez connue dans le monde Android et ne prête pas à explication. Nous utilisons un ShardedPreference pour persister cette information et la renvoyer dans le futur.

Cela donne le code suivant:

```
public class MyApplication extends Application {
*... Other code of the Application object ...*
// Here we are because we receive either the boot completed event
// either the connection changed event
// either the wifi state changed event
ConnectivityManager cm = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
NetworkInfo networkInfo = cm.getActiveNetworkInfo();
// update the preferences
SharedPreferences prefs = getSharedPreferences(MyApplication.CONNECTIVITY_STATUS, Context.MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();
if (null == networkInfo) {
        // This is the airplane mode
        isConnected = false;
        isWifi = false;
        telephonyType = 0;
} else {
        switch (networkInfo.getType()) {
        case ConnectivityManager.TYPE_WIFI:
                if (isConnected == false) {
```

```
                        // then it means the connectivity is back
                        // so we need to notify about that
                        notifyConnIsBackListeners = true;
                }
                isConnected = true;
                isWifi = true;
                break;
        case ConnectivityManager.TYPE_MOBILE:
                if (isConnected == false) {
                // then it means the connectivity is back
                // so we need to notify about that
                        notifyConnIsBackListeners = true;
                }
                isConnected = true;
                telephonyType = telephonyManager.getNetworkType();
                // For information TelephonyType is one of the following
                // switch (telephonyManager.getNetworkType()) {
                // case TelephonyManager.NETWORK_TYPE_LTE:// 150Mb/s
                // case TelephonyManager.NETWORK_TYPE_HSDPA:// 42Mb/s
                // break;
                // case TelephonyManager.NETWORK_TYPE_EDGE:// 215kb/s
                // break;
                // case TelephonyManager.NETWORK_TYPE_GPRS:// 45kb/s
                // break;
                // default:
                // break;
                // }
                break;
        default:
                break;
        }
}
// Store the state
editor.putInt(getString(R.string.telephonyType), telephonyType);
editor.putBoolean(getString(R.string.has_network), isConnected);
editor.putBoolean(getString(R.string.networkStateInitialized), true);
editor.putBoolean(getString(R.string.has_wifi), isWifi);
editor.commit();
if (notifyConnIsBackListeners) {
        notifyConnectivityIsBack();
}
Log.e("MyApplication", "manageConnectivtyState called and return isConnected=" + isConnected + ", isWifi="
                + isWifi + ", telephonyType=" + telephonyType);
}
```

2.2.2 Ventilation de l'information

Nous utilisons le principe de Design Pattern appelé Observer. C'est le principe des listeners.

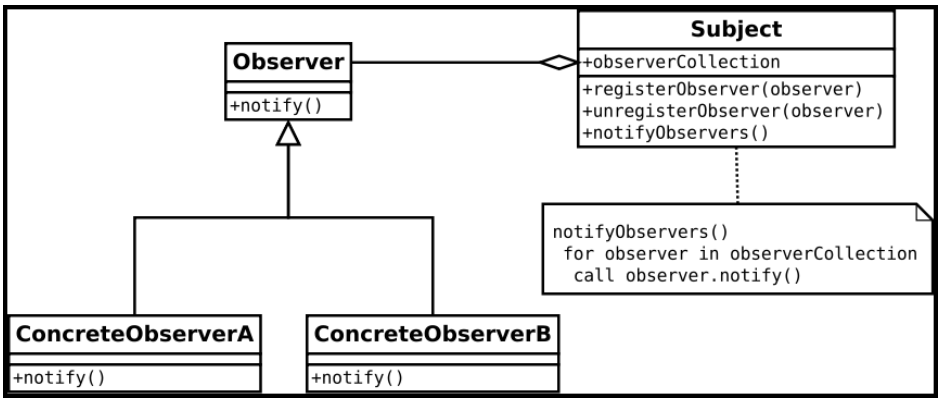

Figure 3: Observer Design Pattern

Pour cela nous avons besoin d'une interface (l'Observer) qui sera implémentée par tous les objets souhaitant être notifiés d'un changement de connectivité réseau :

```
/**
 * @author Mathias Seguy (Android2EE)
 * @goals
 * This class aims to be instanciate by those who want to be notify when the connectivity is back
 */
public interface ConnectivityIsBackIntf {
	/**
	 * CallBack to be notify when the connectivity is back
	 * @param isWifi Tell you if the Wifi is on
	 * @param telephonyType If isWifi==false, it means you'are connected using GPRS and it give you your connectivity type (LTE,Edge,GPRS,HSDPA)
	 */
	public void connectivityIsBack(boolean isWifi, int telephonyType);
}
```

Ensuite, l'objet Application (le Sujet) implémente simplement l'ajout, la suppression de tels éléments et la notification de ceux-ci:

```
public class MyApplication extends Application {
*... Other code of the Application object ...*
/**
 * Call this method when you want to be notify of the connectivity back
 * It means you were offline and then connectivity is back and you want to be notify of that fact
 * Should be called in onResume in activity or in the constructor for others objects
 *
 * @param conBackListener
 */
public void registerAsConnectivityBackListener(ConnectivityIsBackIntf conBackListener) {
	if (null == connectivityIsBackListeners) {
		connectivityIsBackListeners = new ArrayList<ConnectivityIsBackIntf>();
	}
	connectivityIsBackListeners.add(conBackListener);
}
/**
 * Call this method in onPause if you had called registerAsConnectivityBackListener in onResume
 *
 * @param conBackListener
 */
```

```
public void unregisterAsConnectivityBackListener(ConnectivityIsBackIntf conBackListener) {
	if (null != connectivityIsBackListeners) {
		connectivityIsBackListeners.remove(conBackListener);
		if (connectivityIsBackListeners.size() == 0) {
			connectivityIsBackListeners = null;
		}
	}
}
/**
 * This method is called when we switch from no connectivity to connected to the internet
 */
private void notifyConnectivityIsBack() {
// notify the listeners (if there is some because this method can be called even if no
// activity alived)
if (connectivityIsBackListeners != null) {
	for (ConnectivityIsBackIntf listener : connectivityIsBackListeners) {
		listener.connectivityIsBack(isWifi, telephonyType);
	}
}
// The job is done, go back to false
notifyConnIsBackListeners = false;
}
```

Enfin, il ne reste plus qu'à implémenter cette interface dans les objets qui souhaite être au courant de ce changement de connectivité (le ConcreteObserver):

```
public class MainActivity extends MotherActivity implements ConnectivityIsBackIntf,
		SwipeRefreshLayout.OnRefreshListener {

@Override
protected void onPause() {
	super.onPause();
	MyApplication.instance.unregisterAsConnectivityBackListener(this);
}
@Override
protected void onResume() {
	super.onResume();
	MyApplication.instance.registerAsConnectivityBackListener(this);
	//... other stuff
}

@Override
public void connectivityIsBack(boolean isWifi, int telephonyType) {
// Ok so the connectivity is back, we should load the data
if (!dataLoaded) {
	MyApplication.instance.getServiceManager().getForecastServiceData().getForecast(new ForecastCallBack() {
		@Override
		public void forecastLoaded(List<YahooForcast> forecasts) {
			forecastLoadedReturns(forecasts);
		}
		});
	}
	// else do nothing data already loaded
	// then insure the NoNetwork error message is hidden
	findViewById(R.id.txvNoNetwork).setVisibility(View.GONE);
}
```

Ce principe permet de ventiler l'information à toutes les classes qui en ont besoin sans pour autant les obliger à hériter d'un même objet, il suffit d'implémenter l'interface ConnectivityIsBackIntf.

3 La gestion des services avec le ServiceManager

Deux principes forts s'appliquent au ServiceManager:

- Il doit suivre le cycle de vie de l'application;
- Il est le seul à gérer les services.

Les corollaires sont :

- C'est un singleton; il ne peut y avoir qu'une instance du ServiceManager
- Seul l'objet application possède un pointeur sur lui;
- On accède à cet objet via l'objet Application;
- Seul l'objet Application peut l'instancier;
- Il doit instancier les services à la demande;
- Il doit détruire les services quand il faut; quand l'objet Application le lui demande;
- Il doit détruire les Threads à la destruction des services (ou faire en sorte qu'ils se terminent);

Ces principes s'implémentent de la manière suivante:

3.1 Mise en place du Singleton

Au niveau de l'application, l'unicité du ServiceManager se met (peut se mettre) en place simplement:

```
public class MyApplication extends Application {
	/** blabla */
/** The service manager used to manage the services */
private ServiceManager serviceManager;
/** To know if the service manager already exist */
private boolean servcieManagerAlreadyExist = false;
	/** blabla */

/** Managing ServiceManager ****************************************/

/** * @return the serviceManager */
public final ServiceManager getServiceManager() {
	if (null == serviceManager) {
		serviceManager = new ServiceManager(this);
		servcieManagerAlreadyExist = true;
	}
	return serviceManager;
}

/** * @return true if the ServiceManager is already instantiate */
public final boolean serviceManagerAlreadyExist() {
	return servcieManagerAlreadyExist;
}
```

Au niveau du ServiceManager, il suffit de s'assurer que seule l'application peut instancier cet objet et qu'elle ne peut le faire qu'une seule fois. Le constructeur est le suivant:

```
/**
 * Insure only the Application object can instantiate once this object
 * If not the case throw an Exception
 */
public ServiceManager(MyApplication application) {
```

```
        if (application.serviceManagerAlreadyExist()) {
                throw new ExceptionInInitializerError();
        }
}
```

3.2 Accession aux services

Nous aurions pu mettre en place un callback pour le temps d'instanciation des Services de manière à ne pas pénaliser le Thread UI, mais dans cet exemple, cela aurait économisé le temps d'instanciation d'une classe java normale qui ne nécessite pas d'initialisation... ce qui n'a pas d'intérêt.

Une autre remarque importante est que les services de cette application ne sont pas des Services Android. Il n'y a pas d'intérêt dans ce cas précis. Mais nous y reviendrons dans la conclusion.

Ainsi l'instanciation des services est triviale et s'effectue au moyen de getter :

```
/** * @return the forecastServiceData */
public final ForecastServiceData getForecastServiceData() {
        if (null == forecastServiceData) {
                forecastServiceData = new ForecastServiceData(this);
        }
        return forecastServiceData;
}

/** * @return the ForecastServiceUpdater */
public ForecastServiceUpdater getForecastServiceUpdater() {
        if (forecastServiceUpdater == null) {
                forecastServiceUpdater = new ForecastServiceUpdater(this);
        }
        return forecastServiceUpdater;
}
```

3.3 Destruction du ServiceManager et des services

Le point crucial dans ce Design Pattern est de s'assurer de la destruction du ServiceManager et des services quand l'application se termine. Si ce point n'est pas respecter c'est la porte ouverte aux fuites mémoires et autres drainage de batterie.

Dans, cette application, deux points sont à prendre en compte:

- Les Threads n'ont pas besoin d'être annulées durant leur exécution, ils peuvent finir leur traitement, cela n'est pas pénalisant et il est sûr que le traitement se terminera de lui-même;
- Les services ne sont pas des services Android, il suffit que le ServiceManager libère leur pointeur pour que ceux-ci soient libérer quand les Threads se termineront;
- Les services ne sont pas des services Android, ils ne retiennent donc pas l'Objet Application actif.

Il suffit donc d'implémenter le 1 second pattern et le onLowMemory au sein de l'objet Application.

Le 1 second pattern est un Design Pattern dont l'objectif est de vérifier si l'application possède une activité visible par l'utilisateur. Si ce n'est pas le cas, elle doit détruire les services et restituer la mémoire. Si c'est le cas, elle ne doit rien faire. Cette vérification est effectuée dans la seconde qui suit chaque destruction de chaque activité de l'application.

Ainsi les activités de l'application mettent le booléen de l'objet Application, activityAlive, à true quand elles passent dans leur méthode onStart et à false quand elles passent dans leur méthode onStop. Ce principe peut aussi s'implémenter dans les méthodes onPause et onResume. L'objet Application, quand ce booléen passe à false, lance un thread dans une seconde qui vérifie la valeur de ce booléen. Si celui-ci est encore à false, il faut détruire l'application et donc tous ses services pour restituer la mémoire au système.

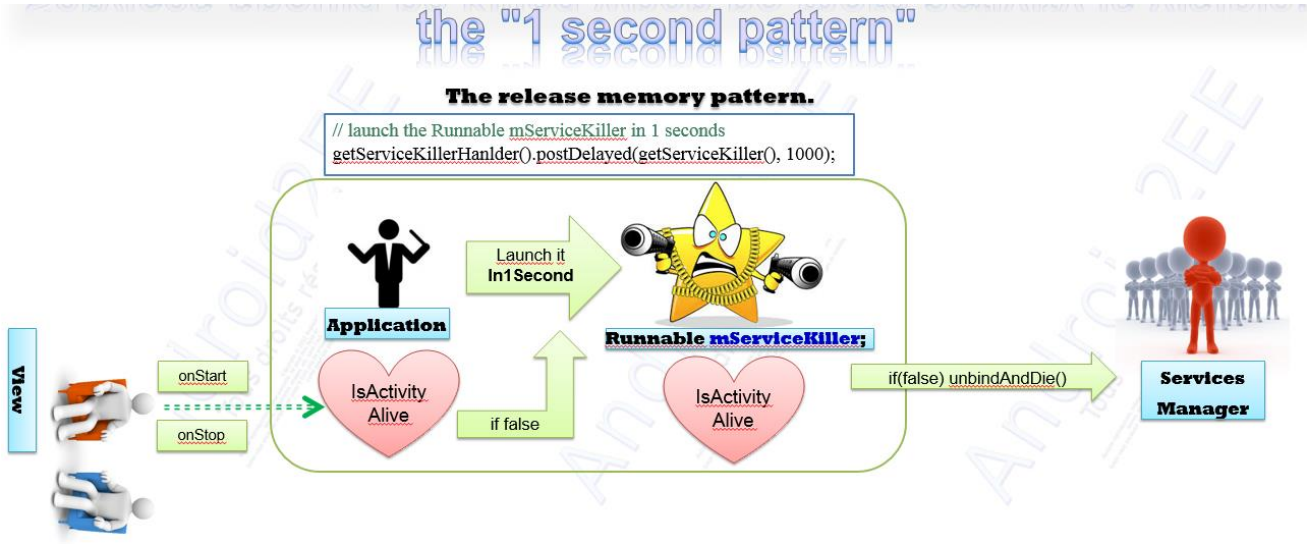

Figure 4: Le one second pattern

De même, lorsque le système appelle la méthode onLowMemory de l'objet Application, il faut que nous détruisions tous les services de manière à libérer la mémoire. Cela n'impactera pas (trop) nos activité, car le lazy loading des services est mis en place.

Ce qui donne:

```
public class MotherActivity extends Activity {

@Override
protected void onResume() {
	super.onResume();
	MyApplication.instance.onResumeActivity();
}

@Override
protected void onPause() {
	super.onPause();
	MyApplication.instance.onPauseActivity();
}
```

et dans l'objet Application:

```
public class MyApplication extends Application {
	/** blabla */
/** Managing destruction **********************/
//Listening for activities life cycle to tricker the serviceManager death
/** * The AtomicBoolean to know if there is an active activity */
private AtomicBoolean isActivityAlive=new AtomicBoolean(false);
/** * To be called by activities when they go in their onStop method */
public void onPauseActivity() {
```

```
		isActivityAlive.set(false);
		// launch the Runnable in 2 seconds
		mServiceKillerHandler.postDelayed(mServiceKiller, 1000);
}
/** * To be called by activities when they go in their onResume method */
public void onResumeActivity() {
		isActivityAlive.set(true);
}
//The 1 second pattern to kill activityManager in 1 second
/** * The Runnable that will look if there are no activity alive and launch the ServiceManager death*/
Runnable mServiceKiller;
/** * The handler that manages the runnable */
Handler mServiceKillerHandler = new Handler() {
		@Override
		public void handleMessage(Message msg) {
			//Kill serviceManager
			killServiceManager();
		}

};
/** * initialize the runnable */
private void initializeServiceKiller() {
		mServiceKiller = new Runnable() {
			@Override
			public void run() {
				if (!isActivityAlive.get()) {
					//one second later still no activity alive
					// so kill ServiceManager
					mServiceKillerHandler
						.dispatchMessage(mServiceKillerHandler.obtainMessage());
				}
			}
		};

}
//Now the Killing method
@Override
public void onLowMemory() {
		super.onLowMemory();
		killServiceManager();
}
/** * Kill the service manager and all the services managed by it */
private void killServiceManager() {
		if (null != serviceManager) {
			serviceManager.unbindAndDie();
			serviceManager=null;
			servcieManagerAlreadyExist=false;
		}
}
```

Enfin, dans le ServiceManager:

```
/**
 * To be called when you need to release all the services
 * Is managed by the MyApplication object in fact
 */
public void unbindAndDie() {
		forecastServiceUpdater = null;
		forecastServiceData = null;
```

}

4 La gestion des données avec le ServiceData et le ServiceUpdate

Cette application possède deux services:

- Le ServiceData qui a à charge la gestion des données; les récupérer en base, les envoyer aux I.H.M. et demander leur mise à jour;
- le ServiceUpdater, lui, doit synchroniser ces données avec le serveur backend.

Les traitements effectués par ces services s'effectuent dans un Thread de background. Dans cette première version du projet, les informations sont renvoyées à l'objet appelant au moyen de CallBak.

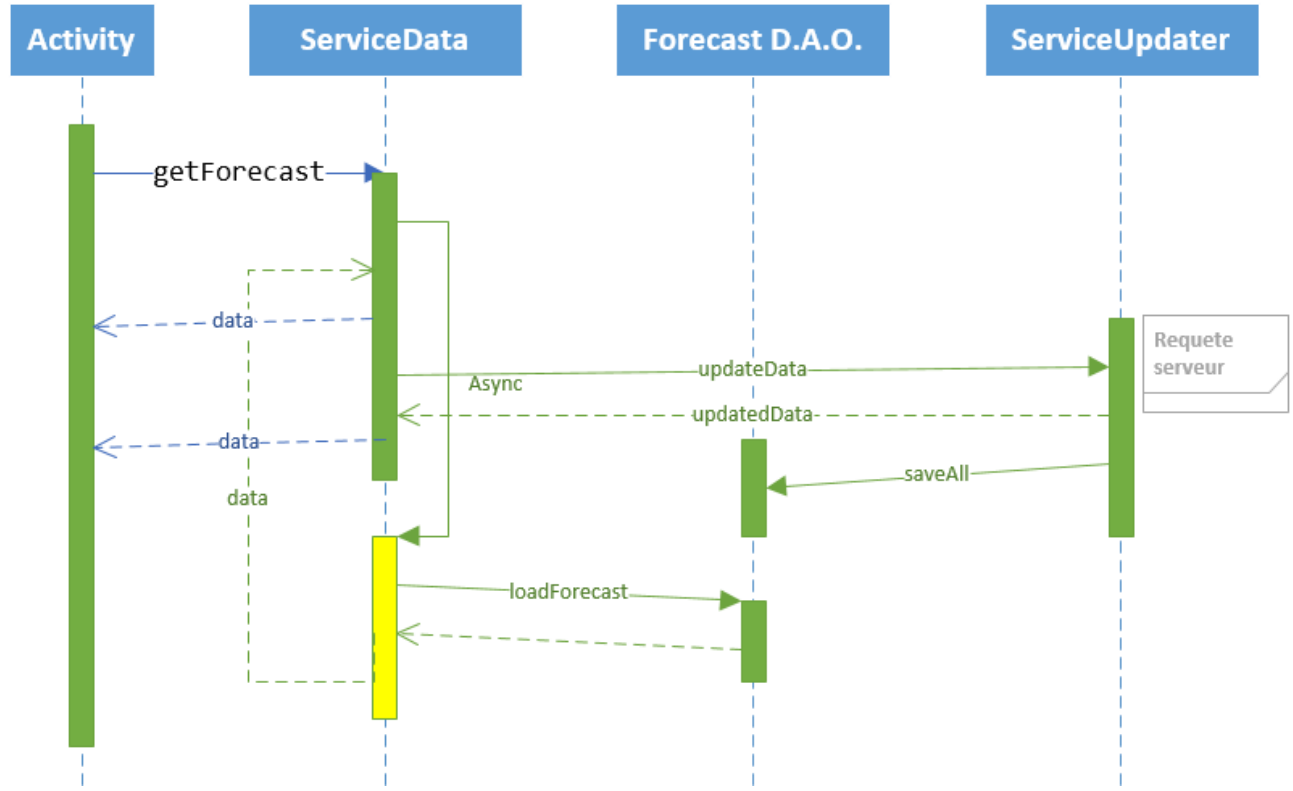

Figure 5: Diagramme de séquence des appels de services

Le code se trouve dans le projet, il n'y a pas grand-chose à rajouter quand on a assimilé cet enchainement.

5 La gestion des Exceptions avec l'ExceptionManager et les ExceptionManaged

La gestion des Exceptions est primordiale dans toutes applications. L'objectif de ce Design est de centraliser leur gestion.

Ainsi, toutes les méthodes qui ont un bloc catch, doivent faire appel à l'ExceptionManager pour centraliser la gestion de l'Exception :

```
catch (ClientProtocolException e) {
```

```
ExceptionManager.manage(new ExceptionManaged(this.getClass(), R.string.protocol_excep, e));}
```

L'idée est de créer une ExceptionManaged et de lui adjoindre des informations qui seront utiles soit pour le débogage soit pour afficher l'erreur à l'utilisateur. Ici, nous ne rajoutons que le message pour l'utilisateur.

L'ExceptionManager peut alors effectuer le même traitement sur toutes les Exceptions. Il doit effectuer deux choses:

- Un feedBack utilisateur;
- Un feedback développeur.

Vous pouvez utiliser Crashlytics ou Accra pour la partie développeur, je ne le présente pas dans le projet. Je me contente de faire un log de l'exception (ce qui n'est pas recommandé).

5.1 Mise en place du feedback utilisateur

Pour mettre en place un feedback utilisateur pertinent et centraliser, deux choses sont effectuées :

- L'ExceptionManager envoie un Intent en lui passant le message d'erreur à afficher;
- La MotherActivity (classe mère de toutes les Activity de l'application) écoute ce type d'Intent et affiche un message à l'utilisateur.

```
public class ExceptionManager {
/** Attribute and constants ****************************************/
/**
 * The name of the Action of the Intent send thought the system to warm the graphical layer
 * (your activity) that an exception occurs and a message as to be displayed to the user
 */
public static final String Error_Intent_ACTION = "EXCEPTION_MANAGER_NEW_EXCEPTION_FIRED";
/**
 * The name of the Extra that handles the error message of the Intent send thought the system to warm the graphical layer
 * (your activity) that an exception occurs and a message as to be displayed to the user
 */
public static final String Error_Intent_MESSAGE = "EXCEPTION_MANAGER_NEW_EXCEPTION_FIRED";
/**
 * The Intent send thought the system to warm the graphical layer
 * (your activity) that an exception occurs and a message as to be displayed to the user
 */
private static final Intent Error_Intent=new Intent(Error_Intent_ACTION);

/** Static method ****************************************************/
/** * @param exception */
public static void manage(ExceptionManaged exception) {
        management(exception);
}
/** * @param exc */
private static void management(ExceptionManaged exc) {
        if (!exc.isManaged()) {
                exc.setManaged(true);
                //Firts fire the error Intent thought the system
                Error_Intent.putExtra(Error_Intent_MESSAGE, exc.getErrorMessage());
                MyApplication.instance.sendBroadcast(Error_Intent);
                // Should prevent the backend server
                //You should do it, This tutorial has no backend$
```

```
                //log
                Log.e("ExceptionManaged", exc.getErrorMessage(),exc);
                Log.e(exc.getRootClass().getSimpleName(), exc.getErrorMessage(),exc);
        }
}
}
```

Et au niveau de l'activité mère, on enregistre le BroadCastReceiver qui écoute cet Intent et il rend visible le panel dans lequel on affiche le message d'erreur. Ce qui donne:

```
public class MotherActivity extends Activity {
@Override
protected void onResume() {
        /** blabla */
        // register the broadcast receiver that listen to the Error_Intent sends by the
        // ExceptionManager
        registerReceiver(receiver, new IntentFilter(ExceptionManager.Error_Intent_ACTION));
}
@Override
protected void onPause() {
                /** blabla */
        // unregister the broadcast receiver that listen to the Error_Intent sends by the
        // ExceptionManager
        unregisterReceiver(receiver);
}
/** Managing the Exceptions ******************/
/** * The txvException */
private TextView txvException;
/** * The click to dismiss string */
private  String strClickToDismiss;
/** * The broadcast receiver that listen to the Error_Intent sends by the ExceptionManager */
private BroadcastReceiver receiver = new BroadcastReceiver() {
        public void onReceive(Context context, Intent intent) {
                displayException(intent.getStringExtra(ExceptionManager.Error_Intent_MESSAGE));
        }
};
/** * Display the Exception carried by the Intent
 * @param String the error message to display */
private void displayException(String errorMessage) {
        // Display the error to your user
        // To do so, every layout of your Activity or Fragment should include the
        // exceptionLayout at it top level
        if(txvException==null) {
                //instantiate the TextView and the Drawable
                        txvException=(TextView) findViewById(R.id.txv_exception_message);
                strClickToDismiss=getString(R.string.txv_exception_clicktodismiss);
                txvException.setOnClickListener(new OnClickListener() {
                        @Override
                        public void onClick(View v) {
                                hideTxvException();
                        }
                });
        }
        if(errorMessage==null) {
                txvException.setText(getString(R.string.no_error_message));
        }else {
                txvException.setText(errorMessage+strClickToDismiss);
        }
        txvException.setVisibility(View.VISIBLE);
```

```
}
/** * Hide the TextView */
private void hideTxvException() {
        txvException.setVisibility(View.GONE);
}
```

Il faut pour que ce Design marche que tous les layouts de l'application incluent le layout d'erreur:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >

   <include layout="@layout/exception_layout" />
   ...reste du layout...
```

6 L'affichage des données avec la MainActivity

L'affichage des données au sein de la MainActivity s'effectue simplement avec une ListView. Elle appelle le ServiceData pour récupérer les données.

7 Conclusion

7.1 Les avantages:

L'architecture implémente les bonnes pratiques Android pour les services R.E.S.T. que nous avons décrit dans l'introduction.

7.2 Les inconvénients:

Je trouve que c'est compliqué pour juste mettre en place l'affichage de la météo récupérée sur Yahoo. Mais c'est le minimum que nous pouvions faire sans faire appel à des librairies.

Le ServiceUpdater n'est pas un service Android. Cela implique que nous ne pouvons pas mettre à jour les données en tâche de fond en laissant le système réveiller ce service, par exemple quand le wifi est actif et l'appareil branché sur le secteur.

Le caching des données au niveau du ServiceData n'est pas exploité.

Les callBack ne sont pas des WeakReferences, ce qui n'est pas optimum au niveau des fuites mémoires et cette communication (Service-Activity) devrait être implémentée par des Intents. Je ne l'ai pas fait car cela complexifie d'autant le projet sans vraiment obtenir un gain majeur.

7.3 Axe d'améliorations:

Les axes d'améliorations sont pour moi les suivants:

- Mise en place d'un bus évènementiel;
- Mise en place d'un O.R.M. (object relational mapping) pour la D.A.O.;
- Mise en place de l'I.O.C. (injection des dépendances);
- Mise en place d'un PoolExecutor pour gérer les Threads et en particulier ceux qui doivent s'arrêter avec l'application;

- Utilisation du caching des données du ServiceData.

8 Android2EE