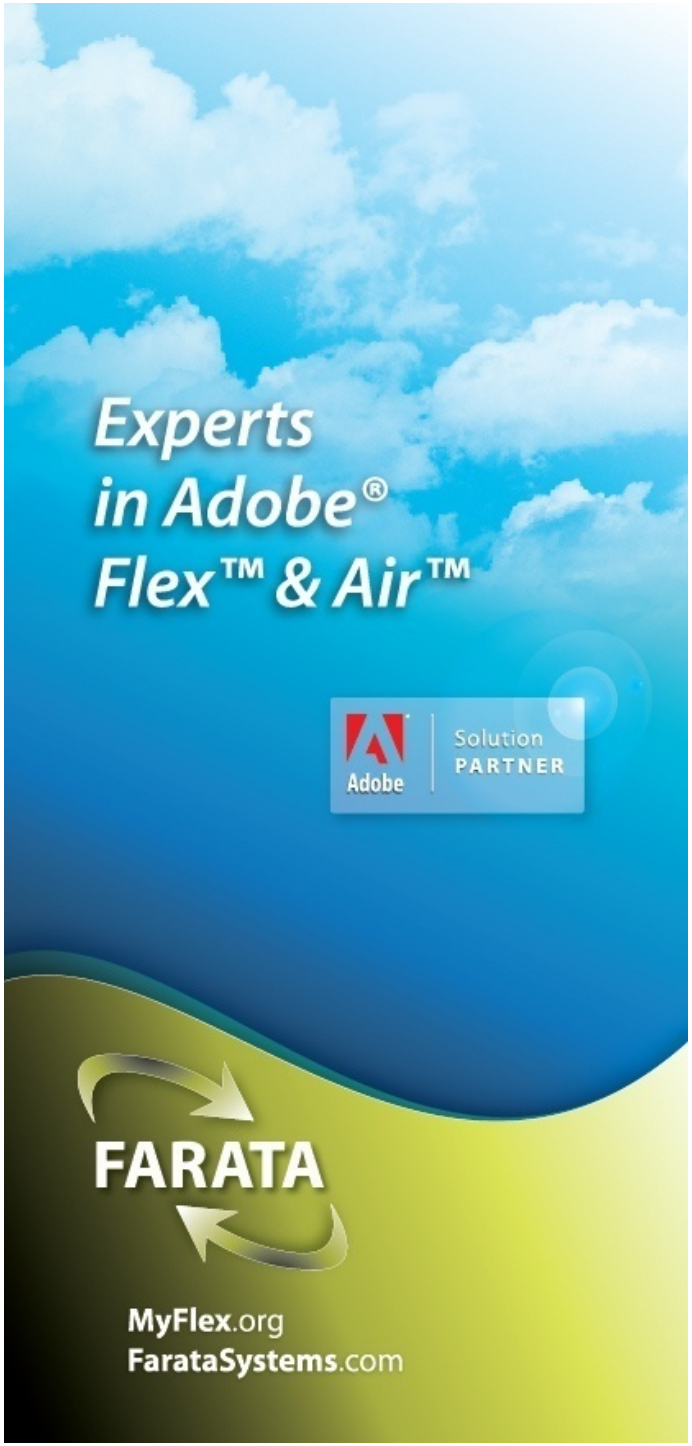



# Development of Flex/BlazeDS Mobile Applications with Introduction To Android and Preview of Clear AAA

Victor Rasputnis,
Farata Systems

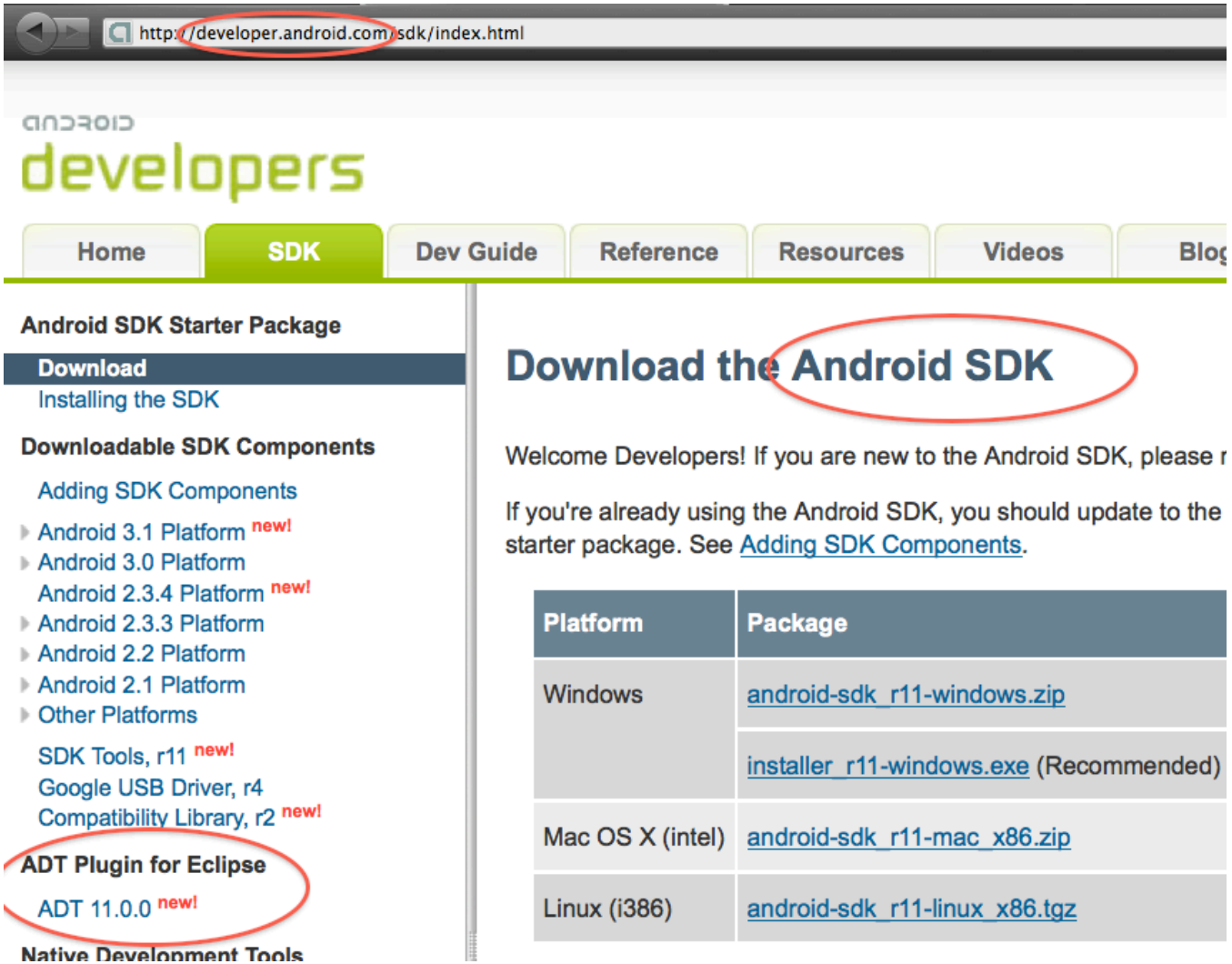
# Why Learn Android?

- In the desktop environment SWF communicates locally with JavaScript/ DOM of the HTML wrapper and remotely with Java/Server object of the Server.
- In mobile device, SWF communicates with …?
- Adobe will be coming up with the native extensions later in 2011. These will take logical place of the ExternalInterface and this is reason #1 to learn Android.
- Farata has developed a flavor of BlazeDS compatible with Android :
  - Your SWF/APK can remote to same-device-BlazeDS and access any Android functions (AMF)
  - Your APK with SWF can execute a request from the SWF anywhere on the net (AMF)
  - Your SWF/APK can still remote to the far-away-remote servers (AMF)
  - Your SWF/APK can serve as "router" of the requests from far-away SWFs to the near SWF (RTMFP)
- And this is #2 reason to learn Android
- All in all Android environment becomes "HTML Wrapper" and "Java Server" circa 2011

# Android Java

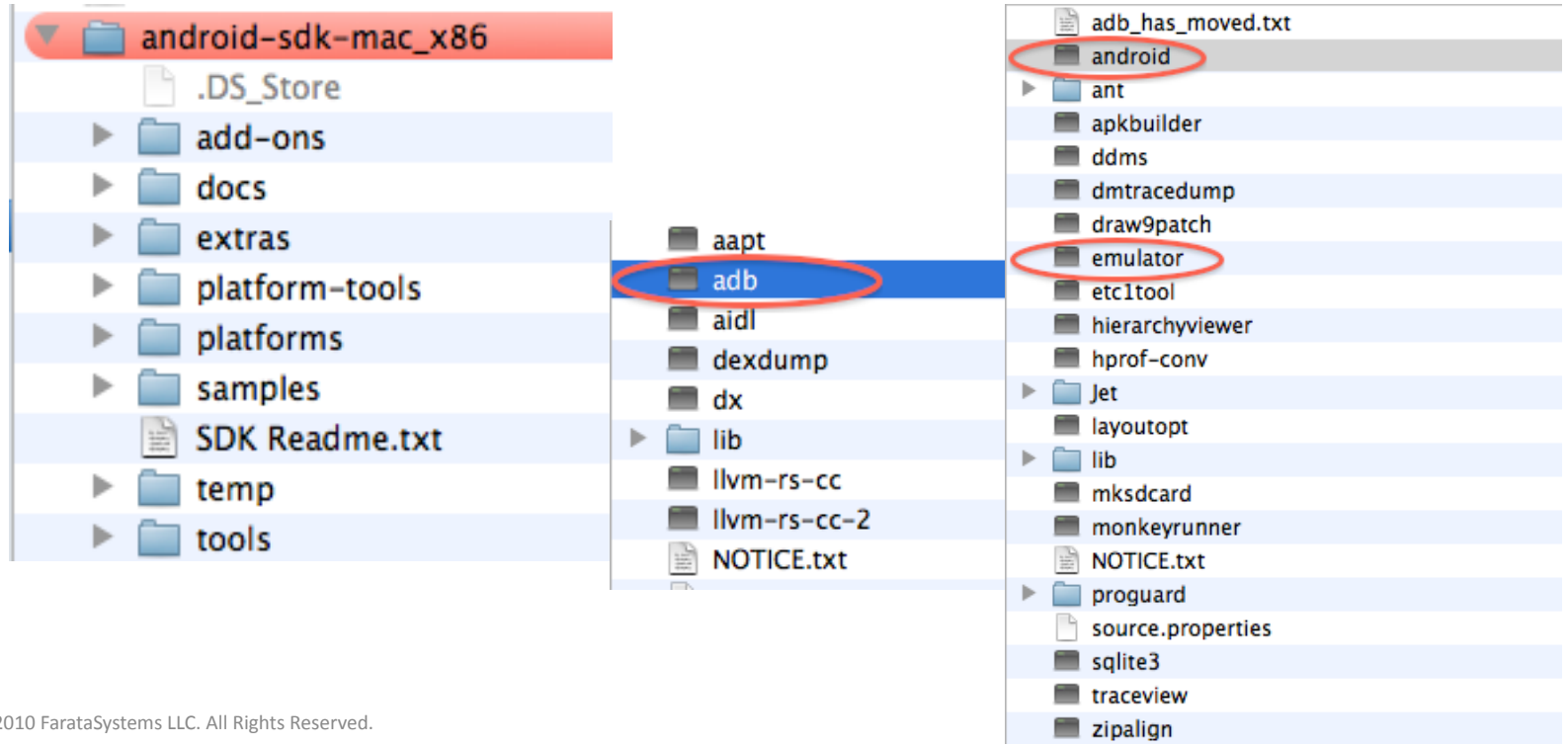
- OS – multi-user Linux. You are not alone in your phone: your phone contacts in SQLLite database are “per account”
- Language – Java, subset of *Apache Harmony*
- VM – *Dalvik*, instead of Sun/Oracle JVM
- Bytecode format – *dex* (Dalvik executable), Dalvic JITs *dex*
- Application installation unit – APK. APK contains *dex*
- Application runs in it’s own VM and it’s own process
- Development environment - Eclipse

# Eclipse plus …

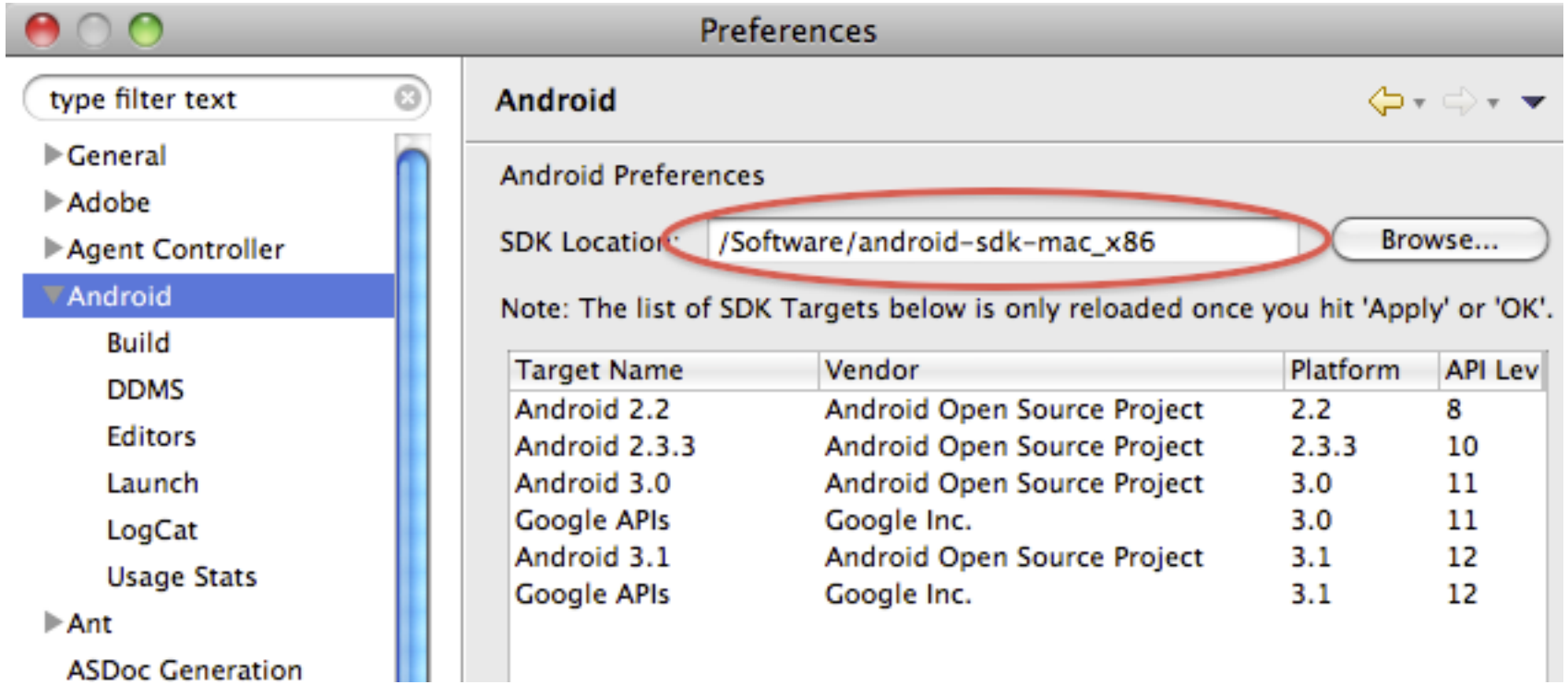


# First Aid Tips

- *adb [un]install [un]*loads APK on device (real or virtual);
- *android* assists in creating virtual devices



# Hook Up SDK to Eclipse

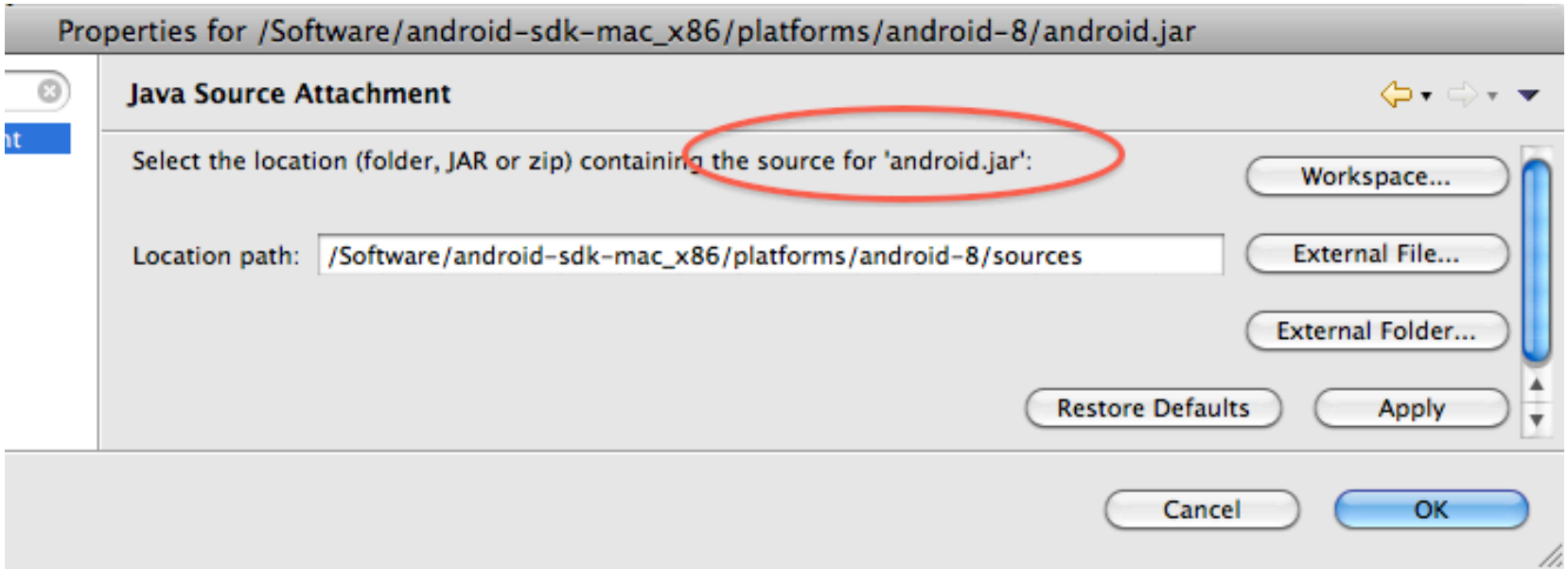


# Source is not an Option ☺

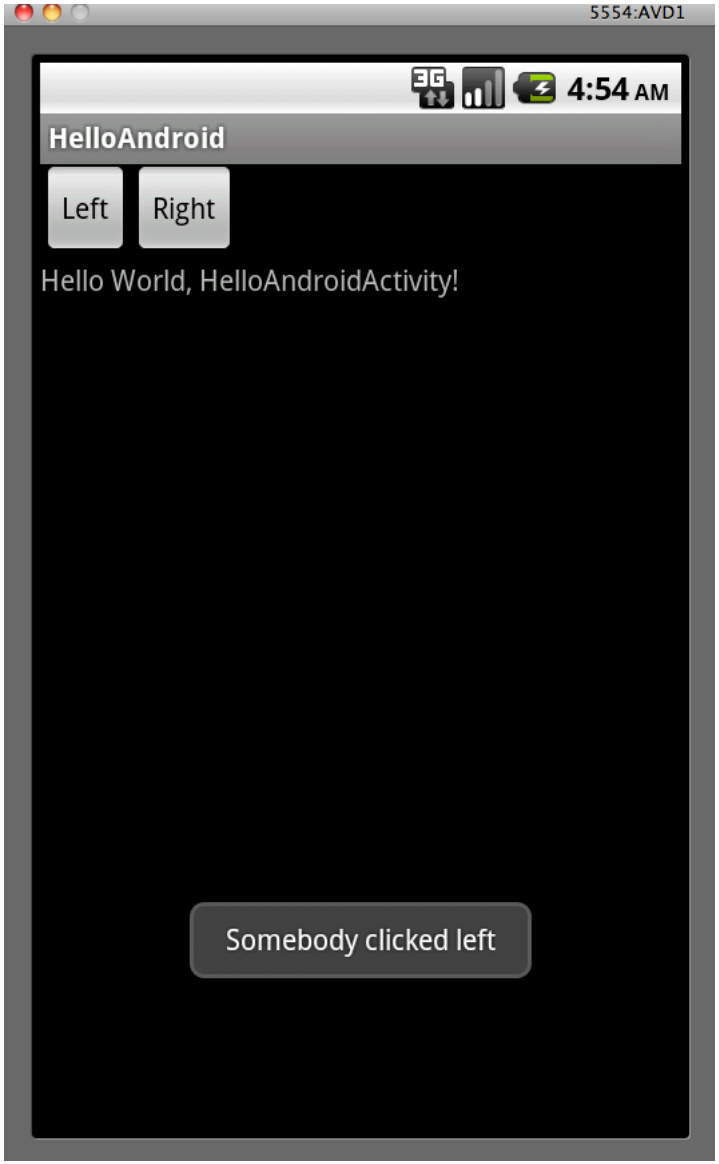
- Get them, even if the only purpose is fighting with extremely slow Helios auto-completion:

http://android.git.kernel.org/?p=platform/frameworks/base.git;a=snapshot;h=froyo;sf=tgz

- Attach source location for android.jar in your project:



# First Words: Activity, Layout, View



# Android HelloWorld

```
public class HelloWorldActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        findViewById(id.btnLeft).setOnClickListener(new OnClickListener() {
                @Override
                public void onClick(View v) {
                    Toast.makeText(getApplication(), "Hello, World!", Toast.LENGTH_LONG).show();
                }
        });
    }
}
```

```
<activity android:name=".activities.HelloWorldActivity" android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
  </activity>
```

# Intent-based Invokation

- To invoke another activity (and not only activity*) – you send message, aka *intent*.
- Routing of the intent is done by Android:
- Explicit routing – when the class name of the target component is known.
- Implicit routing by matching *intent filters* of the component in the *manifest* deployed with the component. Possibilities:

  Match *dataType*: audio/*, match *URI*, match *scheme ( tel:, content:* ), match combination of the above, match *action (*ACTION_CALL, ACTION_EDIT, ACTION_BOOT_COMPLETED)
- If there are multiple components to handle the intent user will get prompted to pick one

# Intent-based Invokation (cont)

- Invoke another activity (from an activity):

```
Intent intent = new Intent();
// Needs <uses-permission android:name="android.permission.CALL_PHONE"/>
//  in your manifest
intent.setAction(Intent.ACTION_CALL);
intent.setData(Uri.parse("tel:17185751433”));
startActivity(intent);
```

- Why com.android.phone.OutgoingCallBroadcaster will be used?

```
<activity android:name="OutgoingCallBroadcaster"
      android:permission="android.permission.CALL_PHONE">
      <intent-filter>
            <action android:name="android.intent.action.CALL"/>
            <category android:name="android.intent.category.DEFAULT"/>
             <data android:scheme="tel"/>
       </intent-filter> . . .
</activity>
```

# Android “Personas”

- *Activity* (“window”), many instances may exists while only one activity (and only one instance) can be in the foreground at a time. Lifecycle: short life through *create-start[*visible*]-resume[*foreground*]-pause-stop-destroy* phases. Even rotation of a device can cause a new life.
- *Service* (“background process”), singleton. Multiple services can be operating in parallel. Lifecycle: between *create-destroy* or *bind-unbind.*

  Non-visual: can provide notifications, but can not do show Toast (alerts)
- *ContentProvider* – “Mini-database”– exposes one or more “tables” to query the data and perform D/U/I requests. Lifecycle: n/a.
- *BroadcastReceiver* – “Stand-alone event listener”. Responds to system or application specific events. (System event that is alive only while processing the intent

# Android Tasks

- Activities in a task form back-stack (LIFO). Task - a collection of activity instances that go between foreground and background, retaining state.
- Task entry activity (aka root activity) that you can start/resume from Home:

```
<activity ... >
   <intent-filter ... >
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
```

- Starting an activity you can use intent flags to force Android to
  - ✓ form a new task (FLAG_ACTIVITY_NEW_TASK)
  - ✓ reuse *existing* activity on top of the stack (FLAG_ACTIVITY_SINGLE_TOP)
  - ✓ reuse *existing* activity anywhere in the stack (FLAG_ACTIVITY_CLEAR_TOP)
- Registering an activity you can use launchMode to force Android to
  - ✓ ensure the *top* of the stack has only single instance (“singleTop”)
  - ✓ all tasks share single instance (“singleTask”), routing - to a different task
  - ✓ reuse *existing* activity anywhere in the stack FLAG_ACTIVITY_CLEAR_TOP

# Android Processes

- Application<==>APK file<==>dedicated process-->separate VM
- Process lives as long as there is a component of the APK, such as Activity, Service, Receiver, ContentProvider that is in use, provided resources are not low. Processes do not get killed at all when resources are abound.
- Type of Processes in priority order:
  - ✓ Host of a *foreground* activity or a service bound to by such an activity (after onResume()), or a service during onCreate()/onDestroy(),

    or a receiver during onReceive()
  - ✓ Host of a *visible* activity (one under a dialog or another transparent FG Activity) or a service bound to by such an activity (after onPause())
  - ✓ Host of a *foreground service* (service that notified user via setForeground())
  - ✓ Host of only *non-visible* activities (after onStop())
- Top priority gets killed last: delegate long-running tasks to services to increase chances to get them complete

# Android Threads

- Primary thread – UI thread. Components, services get created in the same thread: spawning worker threads is your sacred responsibility or freeze to die
- EZ thread spawning for services – extend IntentService instead of Service and override onHandleIntent()
- To make I/O asynchronous from your UI you have to ask nicely, for instance - with AsyncTask:

```
public void onClick(View v) {
  new DownloadImageTask().execute("http://example.com/image.png");
}
private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
    protected Bitmap doInBackground(String... urls) { // this runs in the worker thread
       return loadImageFromNetwork(urls[0]);
    }
    protected void onPostExecute(Bitmap result) {  // this is called with results in UI thread
       mImageView.setImageBitmap(result);
    }
 }
```

# Android Data Access

- Is done with Content Providers
- Android data: audio, video, personal contact information + custom that you build
- Subclass of *android.contentContentProvider* – the only façade to any data in Android
- A contentProvider facades one or many tables with rows and columns, each table corresponds to a public URI, such as:

  *android.provider.Contacts.Phones.CONTENT_URI*
  *android.provider.Contacts.Photos.CONTENT_URI*

  These URI start with "*content://*"
- You do not access a data provider directly: you use ContentResolver instead. The required data provider is determined by Android by URI, which is a required argument in all ContentResolver's methods.

# Using the ContentProvider

- A contentProvider's data is associated with the certain *read permission* and *write permission.* You must declare using one or both in the application manifest, as in

  *<uses-permission android:name="android.permission.READ_CONTACTS"/>*
- To read the data you use ContentResolver.query(uri, …), to manipulate the data you use ContentResolver.insert(uri,…), ContentResolver.update(uri,…), ContentResolver.delete(uri,…)
- If you do not have read permission during *query* or write permissions during *insert* you get *SecurityException*

# Query Android Data

- Use ContentResolver.query() or activity's managedQuery() to get the cursor:

```
final ArrayList<ContactDTO> result = new ArrayList<ContactDTO>();
   final Uri uri = ContactsContract.CommonDataKinds.Phone.CONTENT_URI;
      String[] projection = new String[] {
      ContactsContract.CommonDataKinds.Phone._ID,
      ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME,
      ContactsContract.CommonDataKinds.Phone.NUMBER
};
String selection = ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME + " LIKE 'A%'";
String sortOrder = ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME + " COLLATE
LOCALIZED ASC";
Cursor cursor = managedQuery(uri, projection, selection, null, sortOrder);
```

- Navigate the cursor and populate the data:

```
if (cursor.moveToFirst()) do {
     ContactDTO dto = new ContactDTO();
     dto.setId(cursor.getString(0)); dto.setDisplayName(cursor.getString(1)); …
     result.add(dto);
} while (cursor.moveToNext());
```

# Android Background Processing - Services

- You subclass *Service* or *IntentService* class
- *Started* service runs until expicitly stopped. It can receive commands (you can “start” it multiple times), but it does not return results. Example: socket server
- *Bound* service starts via first *bindService()* and stops with the last *unbindService()*. You can invoke operations on it and get results.
- You can bind to a started service too.
- Service works in the main thread of the app that started it. You create worker threads yourself.

# Building Android Service

- Extend Service, override onStartCommand and/or onBind():

```
public class SampleService extends Service {
@Override
public IBinder onBind(Intent arg) {
  return null; //
}
@Override
 public int onStartCommand(Intent intent, int flags, int startId) {
    Log.i("SampleService", ”Started");
    return START_STICKY; // This service is born to run until explicitly stopped
 }
```

- Manifest registration is mandatory:

```
<application>. . .
      <service android:name="com.farata.samples.services.SampleService">
      </service>
  </application>
```

# Notifications & PendingIntent

- Pending intent – "future" intent with the credentials of the current Activity or current Service

```
Notification notification = new Notification(R.drawable.notificationIcon,
        "Sample service running... ", System.currentTimeMillis()
);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
        new Intent(this, ContactsActivity.class), 0
);
notification.setLatestEventInfo(this,
        getText(R.string.sample_service_running),
        "Select to open Contacts ",
        pendingIntent
);
NotificationManager nm = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
nm.notify(R.string.sample_service_running, notification);
```

# Android “Event-Listeners” aka Broadcast Receivers

- An extension of the BroadcastReceiver, overrides onReceive()
- Types: unordered and ordered
- *Unordered* broadcasts are sent with context.sendBroadcast() and delivered in an uncontrolled order. *Ordered* ones are sent with context.sendOrderedBroadcast() and delivered one at a time per priority of receivers in the relevant manifest.
- Dynamic registration – registerReceiver()/unregisterReceiver, typically in used in activity.onResume()/activity.onPause()
- Static registration - <receiver> tag in the manifest
- If you specify *permission* while sending, receiver’s manifest must have that <uses-permission ../>. If you specify permission while registering receiver, the sender’s manifest must have that <uses-permission…/>

# Using Broadcast Receivers

- This receiver starts ClearDataService ….

```
public class BootReceiver extends BroadcastReceiver{
  @Override
  public void onReceive(Context context, Intent intent) {
      Intent serviceIntent = new Intent();
      serviceIntent.setAction("clear.services.action.ClearDataService");
      context.startService(serviceIntent);
 }
}
```

- … whenever mobile device boots, as per manifest registration:

```
<application>
   <receiver android:name="clear.services.BootReceiver">
      <intent-filter>
            <action android:name="android.intent.action.BOOT_COMPLETED" />
            <category android:name="android.intent.category.HOME" />
      </intent-filter>
  </receiver>
</applictation>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
```

# Clear AAA: BlazeDS for Android

Victor Rasputnis,
Farata Systems

# Quick Intro to Mobile Projects

- Developing Mobile Apps – separate PDF from Adobe
- AirLaunchPad.app (from Tour De Flex or standalone download)
- Navigation:
  - ViewNavigator/TabbedViewNavigator + Views + ViewMenu
  - viewNavigator.pushView(nextView, newData)
  - viewNavigator.popView() + optional override of createReturnObject() + use of viewNavigator.poppedViewReturnedObject within handler of *add*
- Applications on tablets, toys on phones
- (F) Views are not (A) activities, but they get destroyed too (policy)
  - Transactions get very short
  - View – View communication: use *data* property and rely on *add* event U
- Free persistence in Application Storage folder
  - Need to override view serialization methods
- Mobile skins are NOT declarative* (Back to ActionScript ).
  - Multiple DPI

# View Navigator Applications

- Use ViewNavigatorApplication or TabbedViewNavigatorApplication or Application
- (Tabbed)ViewNavigator applications areas: Navigation, Title, Action, Content, Menu
- (Tabbed)ViewNavigator applications maintain their own back stack
- Add view to stack with navigator.pushView(). Remove view from the top of the stack with navigator.popView(). Remove all, but first view with navigator.popToFistView()

```
<s:ViewNavigatorApplication …
firstView="views.CompaniesView"   persistNavigatorState="true”>
  <s:navigationContent>
      <s:Button icon="@Embed('assets/home.png')"  click="navigator.popToFirstView()"/>
  </s:navigationContent>
<s:ViewNavigatorApplication>
```

# View

- View can redefine ActionContent,NavigationContent, TitleContent of the navigator

```
<?xml version="1.0" encoding="utf-8"?>
<s:View …   title="{company.companyName} Employees" >
    <s:actionContent>
      <s:Button label="+" click="editItem(new AssociateDTO(), true) " />
   </s:actionContent>
   …
  <s:viewMenuItems>
      <s:ViewMenuItem label="Delete”  click=“removeAssociate(list.selectedIndex);"/>
      …
 </s:viewMenuItems>
</s:View>
```

# Persistence

- Activated view is given data

```
override public function set data (value:Object):void {
   super.data = value;
   fillAssociates(value as CompanyDTO);
}
```

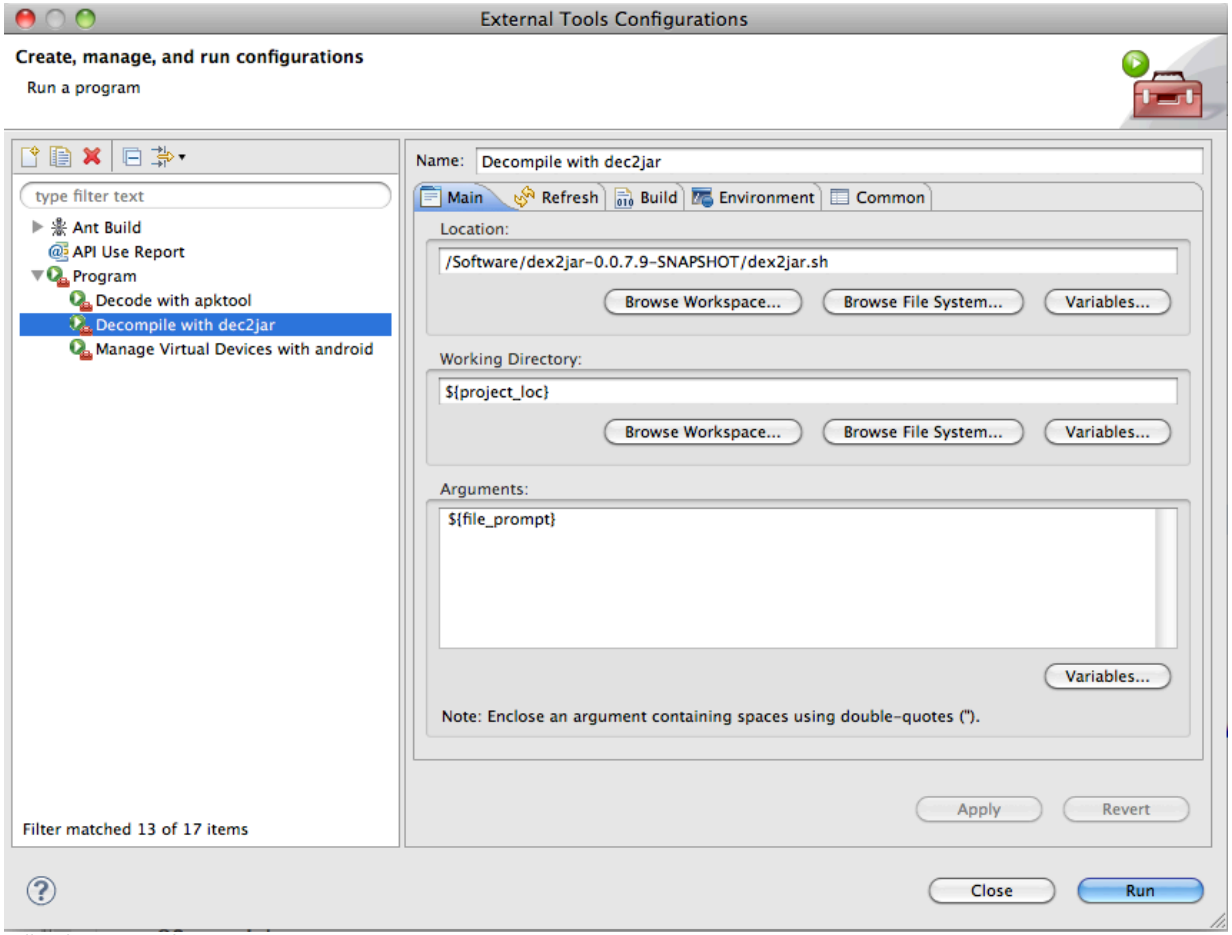
- If application.persistNavigatorState is turned on, the back stack is saved in the application storage area. If the data is not serializable, you have to assist in persisting it:

```
override public function serializeData():Object {
   return {id:company.id, companyName:company.companyName};
}
override public function deserializeData(value:Object):Object {
   var company:CompanyDTO = new CompanyDTO();
   company.id = value.id;
   company.companyName = value.companyName;
   return company;
}
```

# APK->JAR with Dex2Jar

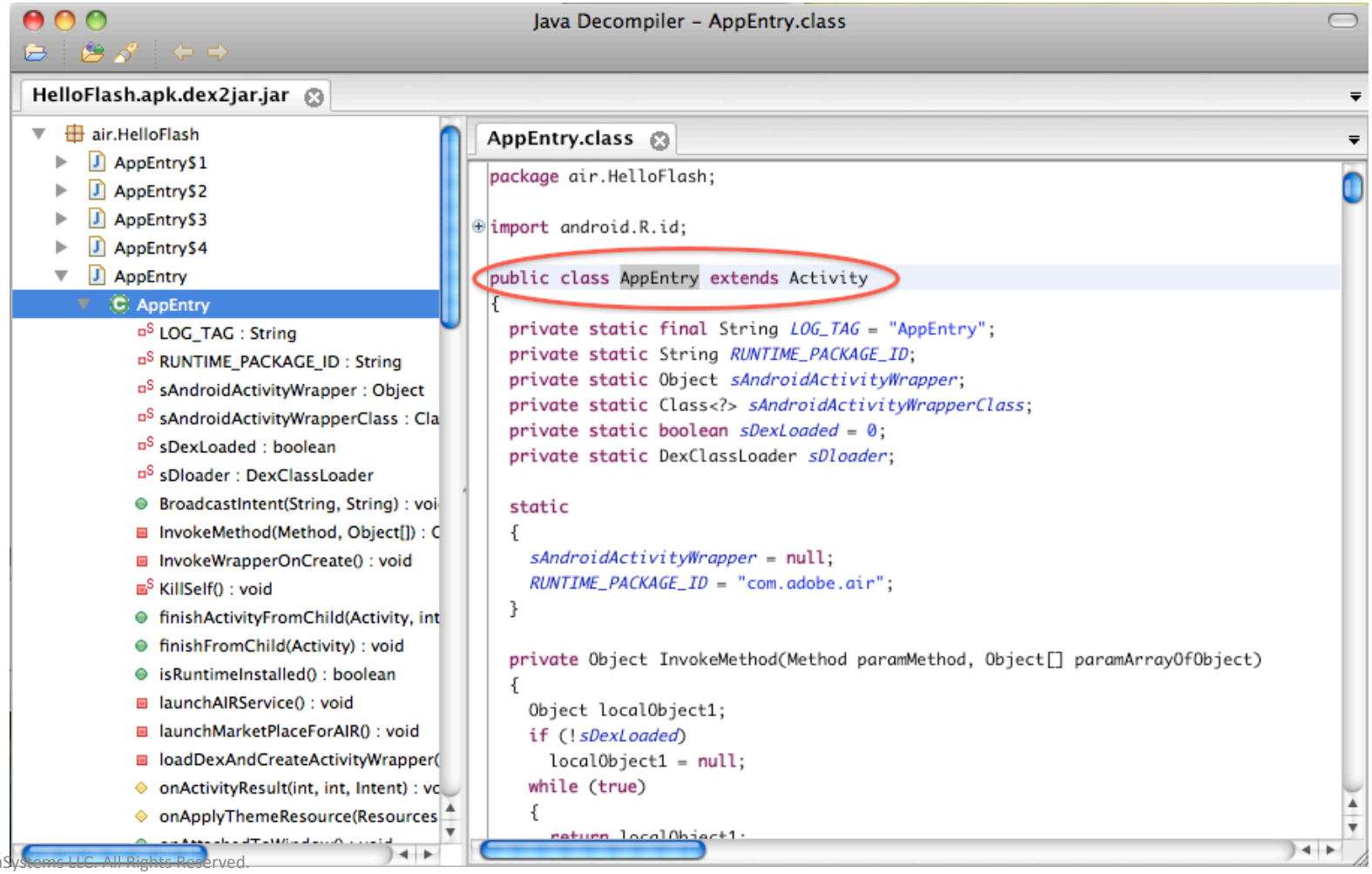
- http://code.google.com/p/dex2jar/

A tool for converting Android's .dex format to Java's .class format



# JAR->Java with JD-GUI

- http://java.decompiler.free.fr/?q=jdgui



# Clear AAA Solution

- Development structure: Java Android Project + Flex Mobile Project.
- Flex Project:
  - Build script that creates ARK, decompiles it to Java and JAR-es out the classes as dependency JAR of the Java Project
- Java Project contains
  - JAR created from the Flex project. It contains activity air.*FlexProjectName*.AppEntry
  - Activity extending the air.*FlexProjectName*.AppEntry
  - Manifest that registers *that* activity as the root activity for the APK
- Also, Java project carries
  - WinStone Servlet Container JAR with extra classes from clear.services.* package
  - BlazeDS JARS with custom implementation of java.beans.*;
  - WEB-INF/flex/services-config.xml + included XML files
- Open Source Eclipse Plugin availability: **October 1st 2011**

  **Clear Air Android Access – Clear AAA,** (clear-triple-a)

# Clear AAA Advantages

- Your Flash-based Android APK becomes a server for the diverse clients:
  - Client application embedded in the APK
  - Client application running anywhere on the net

  Example: A desktop Flex/Air application can remotely invoke an Android application (for signature capture, voice recognition, etc) and receive the result of the action. Likewise, these functions can be invoked from inside the AIR packaged as APK

  This will **not be possible** with upcoming Adobe native extensions.

# Clear AAA API

- AndroidAPIBridge
- LongCalling: Making asyncronous

# LongCalling (Farata Term)

- Problem #1. *Remoting to a  code that needs to be executed on UI thread*.

  BlazeDS provides us with a worker thread which is not the UI thread.

- Problem #2. *Asynchrony on the Java side*.

  Flex Remoting is asynchronous on the client, but it is not ready for async on the server: by the time Android call back is ready, the AMF request/ response exchange is already finished.

# LongCalling Solution: Flex

- LongCall operation must end with “AndWait”, as in “recognizeVoiceAndWait”
- It requires use of the custom RemotingObject supplied by Farata

```
<s:View . . . .
 xmlns:c="library://ns.cleartoolkit.com/flex/clear"
title="Voice Recognition"
>
<fx:Declarations>
      <c:RemoteObject id="service" destination="AndroidJavaDestination"/>
</fx:Declarations>
<fx:Script>
     . . .
    var token: AsyncToken = service.recognizeVoiceAndWait(prompt);
    token.addResponder(new mx.rpc.Responder(
      onRecognizeVoiceResult, onRecognizeVoiceFault
    ));
```

# LongCalling Solution: Java

- LongCall must end with “AndWait”, as in “recognizeVoiceAndWait”
- Code pattern: a transparent wrapper activity that encapsulates synchronous operations like *startActivityForResult()* or *showDialog().* Wrapper finishes on completion of the main operation
- Utility class clear.services.AndroidApiBridge:
  - Invocation of the wrapper activity from a remote Java class should be done with *runActivityAndWait* from
  - Wrapper activity should return results via *complete(originalIntent, results)*, prior to finish()

# LongCall: Java(cont)

```
import static clear.services.AndroidApiBridge.runActivityAndWait;

public List<String> recognizeVoiceAndWait(final String prompt) {
   final Intent intent = new Intent(getApplicationContext(), SpeechRecognitionActivity.class);
   intent.putExtra(RecognizerIntent.EXTRA_PROMPT, prompt);
   return runActivityAndWait(intent, STRING_LIST);
}
final private static Class<List<String>> STRING_LIST = null;
```

```
protected void onStart() {
        super.onStart(); . . .
        startActivityForResult(intent, REQUEST_CODE_RECOGNIZE_SPEECH);
}
@Override
public void onActivityResult(final int requestCode, final int resultCode, final Intent request) {
  if (requestCode == REQUEST_CODE_RECOGNIZE_SPEECH) {
        List<String> result = request.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);
        complete(request, result );
        finish();
   }. . .
}
```

# LongCall Setup

- Custom Remoting Adapter
- Custom Messaging Adapter
- SWC Library to support Custom Remote Object

# Custom Remoting Adapter

- LongCall operation requires use of the custom remoting adapter:

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="remoting-service" class="flex.messaging.services.RemotingService">
. . .
      <adapter-definition id="java-object-long-call"
      class="clear.messaging.adapters.JavaAdapter" default="false" />
      </adapters>
      <destination id="AndroidJavaDestination" adapter="java-object-long-call">
            <properties>
                  <source>com.farata.android.samples.services.RemoteObject</source>
            </properties>
      </destination>
</service>
```

# Custom Messaging Adapter

- LongCall operation requires use of the custom messaging adapter
- LongCall intercepts asynchronous results via internal *longCallResult* destination

```
<?xml version="1.0" encoding="UTF-8"?>
<service id="message-service" class="flex.messaging.services.MessageService">
<adapters>
<adapter-definition id="actionscript"
      class="flex.messaging.services.messaging.adapters.ActionScriptAdapter"
default="true" />
<adapter-definition id="actionscript-long-call"
      class="clear.messaging.adapters.ActionScriptAdapter" />

      <default-channel><channel ref="my-polling-amf" /></default-channels>

      <destination id="longCallResult">
            <adapter ref="actionscript-long-call" />
      </destination>
</service>
```

# Q & A