BWSip Framework - Dev Guide (Android)

Installation

1. Create a new Android project in Android Studio, version 1.0 or greater.
2. Unzip the archive file `bwsip-android-XXXXXXXX-XXXXXX.zip` and copy the `.aar` file to the `libs` folder in your project (if this folder doesn't exist you need to create it). For example, `libs/bwsip-framework.aar`.
3. Include the following lines in your `build.gradle` file:

```
repositories {
    flatDir {
        dirs 'libs'
    }
}

dependencies {
    compile(name: 'bwsip-framework', ext: 'aar')
}
```

4. Edit the `AndroidManifest.xml` file and add the following permissions:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
<uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.VIBRATE" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.WRITE_SETTINGS" />
```

Classes

Main

The *BWSip Framework* has 3 main classes that are used to connect to a registrar and make or receive phone calls. They are:

- **BWPhone:** used to configure the main properties of your phone like the transport used, the STUN servers, set up the SRV resolution, etc.
- **BWAccount:** used to connect to an account in the SIP registrar.
- **BWCall:** used to set and get properties about the calls made/received.

Helper

The *BWSip Framework* has some helper classes that are used to work with codecs, generate tones in the device, get GPS coordinates, etc. They are:

BWCredentials: used to set the user credentials used during the registration. The user can register using the username and password (`BWCredentials.createWithPassword`) or the callsign token (`BWCredentials.createWithToken`). - **BWCodec:** used to list available codecs and change the codec priorities. - **BWTone:** used to play tones in the device. - **BWGps:** used get GPS coordinates. - **BWCallState, BWOutputRoute, BWSipResponse, BWTransport:** list of constants used by the framework.

Getting Started

Basic Steps

In order to use the *BWSip Framework* to make or receive phone calls you need to instantiate the **BWPhone** class. This is the most important class in the framework. The **BWPhone** class is a singleton because there can be only one instance of this object during the entire framework lifecycle.

To create an instance of the **BWPhone** class you can use the code:

```
// Create an instance of the BWPhone class
BWPhone phone = BWPhone.getInstance();
```

With the **BWPhone** object instantiated, you can now set some important properties, like:

```
// The transport type (UDP [default], TCP or TLS)
phone.setTransportType(BWTransport.UDP);

// The log level (from 0 [no log] to 9 [the most detailed log])
phone.setLogLevel(9);
```

You can find more methods and properties available for the **BWPhone** class in the API documentation.

With all properties set in the **BWPhone** object, you must initialize it by calling the method `initialize();`

```
// Initialize the BWPhone object
phone.initialize();
```

Tracking Phone Events

The *BWSip Framework* implements the Delegation Pattern to track phone events, like successful registration, incoming phone calls, changes in the call state, receiving DTMF tones, etc.

The class that will receive these events (the delegated class) must implement the **BWAccountDelegate** and/or **BWCallDelegate** interfaces and pass itself as a parameter to the method `setDelegate` in the **BWAccount** and/or **BWCall** objects; you also need to override the following methods based on the interface that you implement:

BWAccountDelegate

- **onRegStateChanged:** it's triggered when the client authenticates in a registrar, but also whenever the registration status changes. For example: registration failed, registration cancelled, registration updated, etc. Use `BWAccount.getLastState()` to get the last registration state.
- **onIncomingCall:** it's triggered when the client receives a phone call.

BWCallDelegate

- **onCallStateChanged:** it's triggered when the state of a call changes. For example, the call changed from ringing to connected, when the call disconnects, etc.
- **onIncomingDTMF:** it's triggered when the client receives a DTMF tone from the other party.

Example:

```
public class MyActivity extends Activity implements BWAccountDelegate, BWCallDelegate
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        // The MyActivity class -this- will now receive events from BWAccount
        BWAccount account = new BWAccount(phone);
        account.setDelegate(this);

        // The MyActivity class -this- will now receive events from BWCall
        BWCall call = new BWCall(account);
        call.setDelegate(this);
    }

    @Override
    public void onRegStateChanged(BWAccount account)
    {}

    @Override
    public void onIncomingCall(BWCall call)
    {}

    @Override
    public void onCallStateChanged(BWCall call)
    {}

    @Override
    public void onIncomingDTMF(BWCall call, String digits)
    {}
}
```

More information about each method can be found in the API documentation.

Authenticating in a Registrar

In order to make and receive phone calls you must first authenticate in a SIP registrar. This is done through the **BWAccount** class. To use the **BWAccount** class you must instantiate it passing as a parameter the **BWPhone** object that you created previously:

```
// Creating a BWAccount object
BWAccount account = new BWAccount(phone);
```

With the **BWAccount** object created, you can now set the basic properties needed to connect to a registrar:

```
// The registrar URI
account.setRegistrar("registrar.com");

// Set the user credentials using the username and password
account.setCredentials(BWCredentials.createWithPassword("username", "password"));

// Add a proxy to the proxy list (optional)
account.addProxy(proxy);

// Set the user credentials using callsign token
// account.setCredentials(BWCredentials.createWithToken("username", "token"));

// Set the delegate for this account
account.setDelegate(this);

// Attempt the connect to the registrar using the data set above
account.connect();
```

When the registration completes, either successful or as a failure, it will trigger the event **onRegStateChanged** (see Tracking Phone Events).

You can find more methods and properties available for the **BWAccount** class in the API documentation.

Working with Calls

In order to make or receive phone calls you must use the **BWCall** class.

Making a Call

To make a call you must instantiate a **BWCall** object passing as a parameter the **BWAccount** object that you are registered:

```
// Creating a BWCall object
BWCall call = new BWCall(account);
```

With the **BWCall** object created, you can now set the basic properties needed to make a call:

```
// Set the remote URI that will receive the call
call.setRemoteUri("username@registrar.com");

// Set the delegate for this call
call.setDelegate(this);

// Make the call
call.makeCall();
```

Receiving a Call

The client will receive phone calls through the event **onIncomingCall** in the delegated class that you specified previously (see Tracking Phone Events). When the **onIncomingCall** method is called you receive a **BWCall** object as a parameter.

With the **BWCall** object that you received, you must first decide if you want to receive the incoming call and send the proper answer to the client calling you. You can send this answer using the piece of code below according to each scenario:

- If you are already on another call and want to send the busy tone of the client calling you:

```
// Send the busy tone
call.answerCall(BWSipResponse.BUSY_HERE);
```

- If you are available to receive the call you must let the client calling know that the phone is ringing:

```
// Send the ringtone
call.answerCall(BWSipResponse.RINGING);
```

- And finally, to answer the call you send the "OK" signal to establish a connection:

```
// Answer the call
call.answerCall(BWSipResponse.OK);
```

Other Actions

- Inspect the headers on the incoming call INVITE:

```
// returns a Map<String, List<String>> with the headers
call.getInviteHeaders();
```

- Hang-up an active call:

```
// Hang-up the call
call.hangupCall();
```

- Mute or un-mute the call:

```
// Mute the call (the other party cannot hear you)
call.setMute(true);

// Un-mute the call (the other party can now hear you)
call.setMute(false);
```

- Put the call on hold or release the hold:

```
// Put the call on hold (you and the other party cannot hear each other)
call.setOnHold(true);

// Release the hold (you and the other party can now hear each other)
call.setOnHold(false);
```

- Play a DTMF tone on the ring stream with a given volume.
- Notice this function merely plays the tone on STREAM_RING. The focus from the stream should be managed on the application level.

```
tone.playDigit("5", volume);
```

- Play a wav sound on the call stream. The wav must be formatted as 16bit PCM mono/single channel
- Notice this function merely plays the wav on STREAM_VOICE_CALL. The focus from the stream should be managed on the application level.

```
phone.playWav("sampleSound.wav");
```

You can find more methods and properties available for the **BWCall** class in the API documentation.

Objects Lifecycle

The three Main classes in the *BWSip Framework* (**BWPhone**, **BWAccount** and **BWCall**) must be properly deallocated when they are not longer used. You can't rely on Java's garbage collector to remove these objects.

Keep in mind that the main classes are hierarchically related. In other words, if an object is closed then all its children will be automatically closed as well. For example, closing a **BWAccount** object automatically closes all **BWCall** objects created for that account.

In order to deallocate a BWSip object, you must call the `close()` method:

```
// Close the BWCall object
call.close();

// Close the BWAccount object and any BWCall object that belong to that account
account.close();

// Close the BWPhone object and any BWAccount, BWCall object
phone.close();
```

In addition to these classes, you should also dispose of **BWTone** instances in the same way.