Wear Companion Library (WCL)

Wear Companion Library (WCL) is designed to make it easier for developers to work with the Wear ecosystem. It handles a number of routine tasks that any application needs to perform when dealing with the Wear SDK and provides a number of APIs and utility methods to reduce the effort needed there. To measure how much this library could save time, a number of existing sample projects were migrated to use this library and depending on the nature of the sample, the reduction in the size of sample was ranged anywhere from 5% to 40%.

Here is a subset of features that this library provides:

- Handling `GoogleApiClient` set up and lifecycle.
- Providing a `WearableListenerService` that can be activated by adding a small snippet to the client's manifest.
- Allowing Wear applications to make an HTTP request and transparently hand that off to the companion phone app (if connected to one) or making a direct call over WiFi if available.
- Providing a simple API to launch the companion application on any other node.
- Providing, in real-time, the subset of nodes that provide certain capabilities by calling a simple API.
- Sending files from one node to another using simple apis built on top of the `ChannelApi`.
- Sending data over a channel easily by providing `InputStream` and `OutputStream` at the two ends.
- Providing an implementation of an Adapter and Layout for a simple `WearableListView` that can easily show a list of items, with animating circles for the centered item with the possibility to display checked items as well.
- Provides a simple activity and a helper class for recording voice through microphone to a file or streaming that to a different device, and playing back the result.

Introduction

Android Wear SDK contains two separate parts; the first piece is included in the Play Services library (the main piece) and some additional pieces, mostly some UI elements, etc, are distributed as part of a dedicated Wear Support Library. Most app developers use both of these resources to develop an application that is compatible, or runs on, a wear device.

The wear ecosystem is built upon a number of devices (phones, tablet, wear devices) and a cloud component that provides an additional level of connectivity when needed. These devices form a (connected) network of end-points that can share data amongst themselves, send messages to each other or learn about the presence of services offered by other nodes in the network. Due to the distributed nature of this network, any interaction between these nodes is performed in an asynchronous manner. Wear APIs around nodes, messages, data and service discovery fall under this category.

Challenges

Programming to a set of asynchronous APIs usually comes with certain challenges for the developers; this is pronounced if a developer tries to extend her existing (local) application to cover the wear devices; usually the local application is organized and designed in a synchronized manner and adding an asynchronous layer on top of that can prove to be challenging. This, in some cases, can negatively impact a developer who wants to extend her exiting application to interact with a wear device, or at least lengthens the development time.

Another area that developers need to manage on their own is writing boilerplate code to accomplish some routine tasks. Usually at some point in the lifecycle of an application, developer need to connect to the Google API Client for Wear APIs, then register callbacks to learn about the state of connectivity, then bring in additional APIs, such as `Wearable.MessageApi`, `Wearable.DataApi`, etc. to accomplish the needed tasks and has to register more callbacks for each of these, validate that the calls went through, etc. These steps and the associated code is something that will be repeated multiple times, possibly in multiple activities across the same application. Again, this provides more opportunities to make mistakes or to get things wrong.

The third area that developers need to handle when working with wearable application is the specific needs and limitations of the UI components, UX interactions and flow of the application

on a wear device. Not all the components that the UX Design recommends is easily available to developers; they often have to work hard to get things right. In addition, it can result in a non-uniform UI elements that do not behave uniformly across different applications and can cause some serious UX issues for the users.

Objectives

The main goal and objective of WCL is to try to help developers with the abovementioned challenges by providing a wrapper around the SDK that can help managing lifecycle of the application and provide a rich set of easy-to-use lifecycle, data, messaging and discovery callbacks (only the ones that a developer needs at the right time), simplified APIs that cover 80% of the typical use cases and UI components that work out of the box correctly and can be added to a project easily.

Lifecycle and Data, Message and Discovery Callback Management

The entry point to making any communication, sharing of data, services and node discovery is building an instance of the Google API Client for the wearable APIs. This is achieved by first building such instance and then making a connection to the corresponding service. The abovementioned tasks may need to be performed by multiple components (`Activities`, `Services`, etc) and providing a global entity (within the application) to take on the responsibility of holding and managing the Google API Client instance will make it easier to avoid duplication. In WCL, this entity is the **`WearManager`** singleton. This singleton is be accessible from any component in the application, and it needs to be initialized as early as possible. The most appropriate place for this initialization is in the `onCreate()` method of the `Application` instance; this guarantees that initialization has already happened by the time that any component tries to use this singleton.

Sometimes it is important that the application receive messages and callbacks even if the application has not yet been started. This is commonly achieved by implementing a `WearableListenerService` in the application. To this end, WCL provide an implementation of this service, `WclWearableListenerService`, so that it can receive and manage these events even if the application has not yet started. Clients must declare this service in their manifest (see the Initialization and Setup section below). As soon as the client application is launched and is visible, `WearManager` starts this service by calling `Context.startService()`; this is done to ensure that this service is running while `WearManager` is present and is required for the `WearManager` to receive some of its callback events. However, `WearManager` needs to make sure that this service stops when the client application is no longer in front to avoid consuming resources on a wear device when not needed. To accomplish this, `WearManager` automatically registers an application `LifeCycleCallbacks` listener to detect when the application is in the background; when that happens, it stops the `WclWearableListenerService` to avoid having a long-running background service.

The WearManager, upon its initialization, builds an instance of the GoogleApiClient for wearable apis and calls connect() on that instance. When a connection is established, it starts collecting relevant information about the network topology of the connected nodes, and the services that each node provides. This data is cached in the WearManager and by listening to the appropriate events, it keeps this data up-to-date. As a result, clients can immediately obtain the latest data on the connectivity of various nodes, and their capabilities, immediately. The GoogleApiClient callbacks, MessageApi.MessageListener, DataApi.DataListener and CapabilityApi.CapabilityListener are all managed by the WearManager but there might be cases that an application client is also interested in receiving such events. To this end, WCL introduces the WearConsumer interface that contains a long list of callback methods. Since not every component or application is interested in implementing such a long list of callbacks, the AbstractWearConsumer introduces a no-op implementation of this interface so clients can extend this class and only override the subset of callback methods that they are interested in. WearManager provides an easy registration for clients to add any WearConsumer to WearManager, and WearManager, in turn, makes sure to call on these implementations whenever such events are detected.

How to use WCL

To fully take advantage of this library, first it has to be properly initialized. Once that is done, the client applications can access this library to get information, call wear apis or register to receive callbacks with minimal coding of their own.

Initialization and Setup

As was hinted earlier, this singleton must be initialized as early as possible, in the onCreate() method of the Application instance. During initialization, it is required to pass in the (Application) context, and optionally, an array of capabilities that the application wants to dynamically add; in most cases, capabilities are declared in resources, or added locally so the initialization takes the following form:

```
public void onCreate() {
   super.onCreate();
   WearManager.initialize(getApplicationContext());
   ...
}
```

To fully enable the library, its WearableListenerService must also be declared in the manifest:

```
<service android:name="com.google.devrel.wcl.WclWearableListenerService">
  <intent-filter>
    <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
  </intent-filter>
</service>
```

That is all that is needed to enable this library to monitor the changes in the system.

Usage

Any component in the client application can obtain an instance of the WearManager singleton by calling the static method WearManager.getInstance().

1. Registering to receive callbacks

Let's say your application needs to enable a certain UI component only when the Google Api Client connectivity has been established. Moreover, assume the same component also needs to receive any Data that is synced by other nodes. To facilitate this, we need to extend the AbstractWearConsumer and override only the following two methods there:

```
mWearManager = WearManager.getInstance();
mWearConsumer = new AbstractWearConsumer() {
   @Override
   public void onWearableApiConnected() {
      // enable your UI element
   }

   @Override
   public void onWearableDataChanged(DataEventBuffer dataEvents) {
      // handle the data
   }
};

mWearManager.addConsumer(mWearConsumer);
```

Here we are only overriding the two callbacks that we are interested in and not any other callbacks.

***Note**. Make sure that you unregister your consumer when you are done; proper places to register and unregister your consumer in your Activity are in onResume() and onPause() of your activity, respectively (or similar places in other components).*

Consult WearConsumer interface to see the list of available callbacks.

2. Calling Wear related APIs

The WearManager instance also provides an entry to call some of the common APIs (asynchronous or synchronous). For example, to send a message, there are a number of variants that you can call on the WearManager:

```
public void sendMessage(String nodeId, String path, DataMap dataMap,
   ResultCallback<MessageApi.SendMessageResult> callback)
```

Since WearManager handles the Google Api Client internally, you will not see that as an argument for any of the api calls. Here, and in similar calls, you can create and pass an appropriate callback to be used in asynchronous calls, or you can pass null for callbacks. If you do pass a null, WearManager uses a built-in callback that captures the result, and would call the appropriate WearConsumer callback for that api, passing the status code of the result.

Whenever it makes sense, there is also a synchronous version of the same api with the additional "Synchronous" postfix added to the name; since such calls should only be called on a worker thread, a check is done on these apis to throw an IllegalStateException error if they are called on the UI thread.

There is a number of additional methods that provide information about the network (e.g. WearManager.getNodesForCapability(String capability) or WearManager.launchAppOnNode(..)) and some utility methods to make handling of various tasks easier, e.g. WearManager.loadBitmapFromAssetSynchronous(Asset). Consult the JavaDoc for the WearManager and Utils classes in the library to see a complete list.

Examples and Snippets

Here we give some examples on how this library can be used in real scenarios.

1. Sending a message to nodes that provide certain capability

```
WearManager wearManager = WearManager.getInstance();
Set<Node> nodes = wearManager.getNodesForCapability(targetCapability,
      new  SingleNodeFilter(new NearbyFilter());
if (!nodes.isEmpty()) {
   // we have a nearby connected node that provides the desired capability
   Node targetNode = nodes.iterator().next();
   wearManager.sendMessage(targetNode, SOME_PATH, bytes);
}
```

Here, we first obtain a reference to the WearManager singleton. Then we get a list of current nodes that provide the desired capability. Note that the WearManager always has an up-to-date list so you do not need to make any asynchronous call to get the list of nodes. When searching for nodes with a given capability, we may obtain more than one result and often we may want to "filter" the result to a subset that satisfies further criteria. The above method accepts a NodeSelectorFilter object which is a simple interface that the client application can provide. WCL provides a few filters out of the box and the one used in this example is designed to reduce the size of nodes to at most a single node that is also "nearby". If you need a different filtering, you need to provide your own filter. Finally, it sends a message to the selected node.

2. Launching the companion app

Assume you want to launch your app on your watch from your phone. The normal approach could be to identify the watch (using the CapabilityApi) and then sending a message to that node. Then your watch app could have a WearableListenerService to receive that message and launch itself.

If you want to accomplish the same using this library, here is what needs to happen. You have two options: you can use the library to send a specific message on your phone to the designated watch and there, you can receive a specific callback (WearConsumer.onWearableApplicationLaunchRequestReceived()) to react to and launch your activity. There is, however, a simpler alternative approach:

```
On the phone:
WearManager.getInstance().launchAppOnNode(watchActivityName, bundle,
relaunchIfRunning, targetNode);
```

or

```
On the phone:
WearManager.getInstance().launchAppOnNodes(watchActivityName, bundle,
relaunchIfRunning, capabilityName, filter);
```

Here, watchActivityName is the name of the activity on your watch that you want to launch (use a fully qualified name) and bundle can be used to provide additional info to the intent that is used to launch your activity. There are two variations, one assumes you have a target node in mind and the other assumes you want to target based on the capability that your target node provides. The rest is handled by the library! Note that the very same approach works if you want to start your companion app on your watch from your handheld device.

3. Making an HTTP Call from the watch

Sometimes the application running on the wear device requires to reach the internet to obtain or to send some data. The proper approach is to use the paired phone to facilitate this. To provide a robust experience, however, there are times that the watch is not directly connected to a phone. The Wear Companion Library provides a simple set of APIs to handle both cases; if you use this library to make an HTTP call, the library decides whether it can be sent to the phone: if there is a connected phone, the request is automatically sent to the phone and your companion phone app can receive a callback (using a registered WearConsumer) from the library to that effect; your companion app can make the network call and then pass the result back to the library which transfers it back to the watch and hands the result back to the caller on the watch. If the library determines that there is no connected phone and if there is a wifi

connectivity on your watch, then the library makes a direct call and returns the result back to the caller, in a way that the caller doesn't need to distinguish the two cases.

Here is an example of making a simple HTTP GET request:

On the watch, making an HTTP call:

```
// assuming that your phone companion app provides HTTP_HANDLER_CAPABILITY
Set<Node> nodes =
      mWearManager.getNodesForCapability(HTTP_HANDLER_CAPABILITY);
String nodeId = null;
Node node = Utils.filterForNearby(nodes);
if (node != null) {
   nodeId = node.getId();
}
try {
   new WearHttpHelper.Builder(url, MainActivity.this)
       .setHttpMethod(WearHttpHelper.METHOD_GET) // GET is default
       .setTargetNodeId(nodeId)
       .setHttpResponseListener(responseHandler)
       .setTimeout(10000) // default is 15000 ms = 15 seconds
       .build()
       .makeHttpRequest();
} catch (IllegalStateException e) {
   Log.e(TAG, "No Api Client Connection");
} catch (IllegalArgumentException e) {
   Log.e(TAG, "Arguments are missing for the http call", e);
}
```

Here we first find the node that provides the capability of handling HTTP Requests. Then we use the WearHttpHelper class, that is part of WCL, to build and send the request. The response will be handed over to the responseHandler listener:

On the watch:

```
responseHandler = new WearHttpHelper.OnHttpResponseListener() {
   @Override
    public void onHttpResponseReceived(String requestId, int status,
          String response) {
      // check the status of response and handle the response
   }
};
```

At the same time, when there is a connected phone, request will be passed to your companion app so here is what your companion app has to do:

On the phone, register a WearConsumer:

```
mWearConsumer = new AbstractWearConsumer() {
   @Override
   public void onWearableHttpRequestReceived(String url, String method,
         String query, String charset, String nodeId, String requestId) {
      // handle the request, “nodeId” is for the node that had
      // sent the request; it is needed in the response
   }
};
```

After the request is processed by your phone application, it can send the result back:

On the phone:

```
mWearConsumer.sendHttpResponse(response, statusCode, nodeId, requestId);
```

If needed, you can track the response and match it against the original request using the “requestId”.

4. Receiving callbacks even if the app is not running

This library provides a WearableListenerService component that needs to be registered in the manifest (see the section on Initialization and Set Up). This service is invoked when any of a number of events is received by the framework for the given application, such as when a new message is arrived, or shared data is changed, etc. The easiest approach to receive such events and callbacks is to register a WearConsumer listener in the Application instance of your application (right after initializing the WCL) and override the methods you are interested in. For example, to receive a callback when the shared data is changed, do the following in your Application instance:

In your Application class:

```
@Override
public void onCreate() {
   super.onCreate();
   WearManager.initialize(getApplicationContext());
   final WearManager wearManager = WearManager.getInstance();
   WearConsumer wearConsumer = new AbstractWearConsumer() {
      @Override
      public void onWearableDataChanged(DataEventBuffer dataEvents) {
         // handle the dataEvents as usual
      }
   };
   wearManager.addWearConsumer(wearConsumer);
```

```
}
```

5. Sending a file from one node to another, using the ChannelApi

Sending a file from one node to another is a very straightforward task using the WearFileTransfer class provided by this library. Here is a snippet on the sender side:

```
WearFileTransfer fileTransfer = new WearFileTransfer.Builder(targetNode)
        .setTargetName(targetName)
        .setFile(fileToBeSent))
        .build();
fileTransfer.startTransfer();
```

Here, we build an instance of WearFileTransfer by specifying the targetNode (where to send the file), the targetName (what the file should be named at the receiving end) and a reference to the file that needs to be transferred. Then we simply call the startTransfer() method to initiate the transfer and that is all that needs to be done. Behind the scene, this class opens a channel with a very specific path. Then it calls Channel.sendFile() method to send the file. On the receiving end, the WearManager will be notified when a channel is opened and by inspecting the path, it recognizes that a file transfer is happening so it handles the rest using Channel.receiveFile() and saves the file in the (private) application data storage under the requested targetName. WearManager also calls WearConsumer.onWearableFileReceivedResult() to inform the receiving client that a file transfer has just happened with additional information so that the clients can learn about the status of file transmission and retrieve the location for the saved file.

6. Low-level I/O Over a Channel

WearFileTransfer also provides a low-level mode to stream data between two nodes. The sender side needs to request an OutputStream and when it receives one (through the WearFileTransfer.OnChannelOutputStreamListener interface), it can start writing to this OutputStream. On the receiver side, clients can register to listen to WearManager.onWearableInputStreamForChannelOpened(statusCode, requestId, channel, InputStream) and when called, they will have access to the channel that was opened and an InputStream to read from. Here is a snippet on the sender side:

```
WearFileTransfer fileTransfer = new WearFileTransfer.Builder(targetNode)
        .setOnChannelOutputStreamListener(listener)
        .build();
fileTransfer.requestOutputStream();
```

The WearFileTransfer.OnChannelOutputStreamListener listener will give the sender access to an OutputStream, so the sender can start writing to the stream. On the receiver side, register a WearConsumer and override the following method to get a hold of InputStream for your consumption:

```
mWearConsumer = new AbstractWearConsumer() {

   @Override
   public void onWearableInputStreamForChannelOpened(int statusCode, String
requestId, Channel channel, InputStream inputStream) {
      if (statusCode != WearableStatusCodes.SUCCESS) {
         // failed, do as needed!
         return;
      }
      // success, consume the bytes coming through the inputStream!
      ...
   }
}
```

7. Building a WearableListView easily

In many cases, developers need to present a list of items to the user to allow them select one. In these cases, one can use a WearableListView and implement an adapter and add a layout. For the typical behavior common in Google apps where the centered item stands out by using a larger icon, developers would need to dig deeper and add animations themselves. Since this is a common pattern, WCL provides most of the required plumbing for the user to present a list of strings where user can select one:

To create this UI, here is a snippet:

```
String[] data = new String[]{"Option 1", "Option 2", "Option 3"};
WearableListConfig config = new WearableListConfig.Builder(data)
      .setCheckedIndex(1) // index starts from 0
      .setRequestCode(REQUEST_CODE)
      .setIcon(R.drawable.ic_person_24dp)
      .setCheckable(true) // we declare that we can have "checked" state
      .setHeader("Options") // optional header
      .build();

WearManager.getInstance().showWearableList(activity, config);
```

Here, `data` holds a String array of the items in the list. This snippet starts a new activity that is part of the library and when a selection is made, that activity will close and brings you back to the caller activity. To receive the information about the user's selection, you need to override onActivityResult(); a wrapper class, `WearableListConfig.WearableListResult`, is included in the library to ease the handling of the response:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
   if(resultCode == RESULT_OK){
      WearableListConfig.WearableListResult result
            = new WearableListConfig.WearableListResult(data,
                  requestCode);
      if (result.isHandled()) {
         Log.d(TAG, String.format("Position: %d, Value: %s",
             result.getSelectedIndex(), result.getSelectedValue()));
      } else {
         Log.d(TAG, "Another activity is returning a result here ...");
      }
   }
}
```

Remember to declare this built-in activity in the application's manifest:

```
<activity
 android:name="com.google.devrel.wcl.widgets.list.WclWearableListViewActivity">
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
   </intent-filter>
</activity>
```

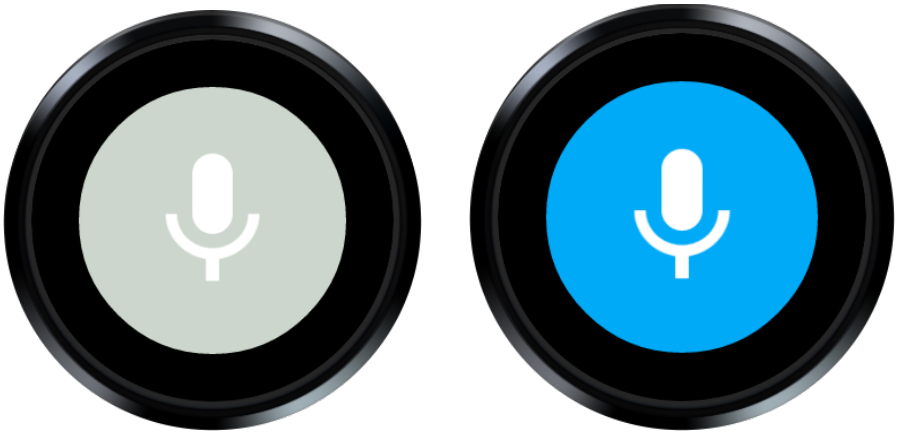
You can also embed a WearableListView in your application/layout. In order to do that, you would need to extend the abstract activity `AbstractWearableListViewActivity` and implement two methods there as outlined in the following snippet:

```
public class MyListActivity extends AbstractWearableListViewActivity {

   @Override // AbstractWearableListViewActivity
   public void onItemClicked(int position, String value) {
      Log.d(TAG, "Item clicked: position=" + position
          + ", value=" + value);
      ...
   }

   @Override // AbstractWearableListViewActivity
   public WearableListConfig getConfiguration() {
      String[] data = new String[]{"Option 1", "Option 2", ...};
```

```
        return new WearableListConfig.Builder(data)
                .setAmbient(true)
                .setHeader("Options")
                .build();
    }
}
```

8. Recording Voice

WCL provides a dedicated activity, WclRecorderActivity, and a helper class, WclSoundManager, to ease the process of recording voice on a wear device. The WclRecorderActivity, when called, opens a full screen activity with an inactive microphone image; when user taps on that, recording starts and continues till user taps on that again:



At that point, this activity will close itself and application is sent back to the caller activity. Here is a snippet that shows how to start this activity:

```
Intent intent = new Intent(this, WclRecorderActivity.class);
intent.putExtra(WclRecorderActivity.EXTRA_RECORDING_FILE_NAME, "audio_file");
startActivityForResult(intent, RECORDING_REQUEST_CODE);
```

The above snippet writes the result of recording into a file named “audio_file” in the application’s private data storage. When finished, the caller activity will receive a callback in onActivityResult() which contains an intent with the information about the success or failure of the recording (see the JavaDoc for the details).

It is also possible to stream the output of the microphone to another node, for, say further processing. To achieve that, use the following snippet:

```
Intent intent = new Intent(this, WclRecorderActivity.class);
```

```
intent.putExtra(WclRecorderActivity.EXTRA_STREAMING, true);
intent.putExtra(WclRecorderActivity.EXTRA_NODE_CAPABILITY,
        VOICE_PROCESSING_CAPABILITY);
startActivityForResult(intent, RECORDING_REQUEST_CODE);
```

Here, we are streaming the output of the microphone directly to a node that provides the VOICE_PROCESSING_CAPABILITY (it is possible to directly set the target node through its nodeId). On the target node, the onWearableInputStreamForChannelOpened() callback will inform the application that an InputStream is available to receive the streaming data.

WclSoundManager is a helper class that is used by this activity and provides a number of apis to record or play a recorded (or streaming) voice file; those apis can be used independently if needed. For details, please consult its JavaDocs.

Appendix

Android Studio Setup

In order to use this library in Android studio, you can follow one of the following two options:

1. Build and use the library as an AAR package. If you decide to use the compiled and packaged version of the library, all you would need to do is to add this library, in aar format, into your dependency for Wearable and Application modules of your project. To accomplish this, checkout the library and build the package:
    a. Build the AAR package by running ./gradlew build. The result will be an AAR package that can be found in WCL/build/outputs/aar/ directory of the project.
    b. Create a libs/ directory inside your project's root directory and place WCL-debug.aar there.
    c. Update the build.gradle in your Wearable and Application modules and add the new libs directory as a repository:

```
repositories {
  jcenter()
  flatDir {
     dirs '../libs'
  }
}
```

    c. Add the WCL-debug.aar to the dependencies:

```
dependencies {
  ...
  compile 'com.google.devrel.wcl:WCL-debug@aar'
}
```

2. An alternative approach is to use the source for this library and add that to your project as a module. Follow these steps:

   a. Checkout the WearCompanionLibrary project and create a symbolic link to it parallel to your project (i.e. your project and WearCompanionLibrary should share the same parent)

   b. In the root of your project, edit the file “settings.gradle” and add the highlighted phrase:

```
include ':Wearable', ':Application', '..:WearCompanionLibrary:WCL'
```

   c. Edit build.gradle in both Wearable and Application modules and add the highlighted dependency:

```
dependencies {
  ...
  compile project(':..:WearCompanionLibrary:WCL')
}
```

   Then you will see a new module in your Android Studio, called WCL, that is the source for the library.

Sample Code

A sample app, WclDemoSample, has been created and open sourced to give you an opportunity to see how this library can be used in a real settings.