linkalist SDK for Android

Displaying a List

30 November 2018


# Introduction

So, now that you’ve got your linkalist app starting up and allowing your customers to log in, the next step is to make it display a list so that it can do something useful. Our first tutorial is going to help you get set up with a simple list connected to your data.

# Setting Up

The first step is to modify your activity’s layout file so that it uses the library’s “SimpleListFragment” rather than the one created when you created the activity. To do this, edit your layout and modify the fragment’s name to be io.linkalist.linkalist\_client.fragments.SimpleListFragment.

<FrameLayout android:id="@+id/fragment\_container"
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 android:layout\_width="match\_parent"
 android:layout\_height="match\_parent"
 tools:context="mobi.divelist.app.DiveListActivity">
 <fragment
 android:id="@+id/fragment\_instance"
 android:name="io.linkalist.linkalist\_client.fragments.SimpleListFragment"
 android:layout\_width="match\_parent"
 android:layout\_height="match\_parent"/>
</FrameLayout>

Your app should now start and run although it won’t do anything useful yet. The next step is to connect it up to a list in your linkalist application. To do this, you need to first enable the list for API use. As the SDK operates on views, this is done by going to the Build tab of your app and editing the view’s API setting. For now, all you need to do is tick the “On Webservice” option.



Next, you need to modify your start up activity so that it requests the application details. This is used to allow the linkalist client SDK initialise itself with all of the list details of your application. This is best done at the point after you check the login details.

if (false == LinkalistCore.*getInstance*().isLoggedIn())
{
 String titleText = getString(R.string.*login\_title*);
 LoginActivity.*launchActivityWithIcon*(this, R.drawable.*ic\_divelink\_flag*, titleText);
}
else
{
 LinkalistCore.*requestApplicationDetails*(this);
}

To make this work you’ll also need follow the hint and let your main activity implement “DetailsResquester”. This will prompt you to add two functions to your application, which you should do.

public class DiveListActivity extends AppCompatActivity implements DetailsRequester

Now, you need to check this works cleanly, so add a query into detailsCompleted to verify that the list has been found. Then add a breakpoint after the query to inspect the resulting list. If this is null, something has gone wrong – most likely you’ve got the wrong list name or you’ve not set up the view to be visible to the api.

@Override
public void detailsCompleted()
{
 List test = LinkalistCore.*getInstance*().findListByName("Your Dives");
}

You can inspect what is available to the api, by navigating to your “Build” page and looking at the documentation. Any available views will be listed here. The findListByName query is case-insensitive. Anyway, after that test works, it’s now time to connect up a basic list request. You need to replace the detailsCompleted function with what’s below and you should be good to go so long as your original view has a name column defined. You will need to change the name called in setListName with that of your own.

@Override
public void detailsCompleted()
{
 FragmentManager fm = getSupportFragmentManager();
 SimpleListFragment fragment = (SimpleListFragment) fm.findFragmentById(R.id.*fragment\_instance*);
 if (null != fragment)
 {
 fragment.setListName("Your Dives");
 fragment.initialise();
 }
}

That will get you up and running with a very simple and boring list that will do the job. However, this won’t look particularly nice and needs some more work so that you can apply your own design skills.



# Formatting your list

The first thing you’ll need to do is start a new XML layout. We’ll just set up a very simple one to get started

<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout\_width="match\_parent"
 android:layout\_height="60dp"
 android:padding="@dimen/standard\_medium\_margin">
 <TextView
 android:id="@+id/dive\_name"
 android:layout\_width="match\_parent"
 android:layout\_height="match\_parent"
 android:gravity="center\_vertical"
 android:textColor="#FF0000"/>
</android.support.constraint.ConstraintLayout>

Now we need to set up the mappings in our code so that the SDK knows which data to map onto each field. So assuming we’ve got a name field in our data, we need to write the following bit of code in our activity.

@Override
public void detailsCompleted()
{
 FragmentManager fm = getSupportFragmentManager();
 SimpleListFragment fragment = (SimpleListFragment) fm.findFragmentById(R.id.*fragment\_instance*);
 if (null != fragment)
 {
 fragment.setListName("Your Dives");
 fragment.setDefaultCellLayout(R.layout.*list\_item\_dive*);
 fragment.setMapping(R.id.*dive\_name*, new MappingData("name", MappingData.MappingType.*TEXT*));
 fragment.initialise();
 }
}

This allows you to set up a mapping that will put data from the “name” filed in the data into a text view with an ID of “dive\_name”. We have mappings defined for Text, Images and Switches. The last shows or hides an image (or text) depending on the value of a Boolean. After setting up this code and layout, you should end up with your app looking something like this.

## More Columns

From this point, it is a relatively simple matter to include more columns. Any column in your is available appear so long as it is not flagged as “Item Only”. You can make “Item Only” columns appear in your list if you tag them “Api Lists” in your view setup. In any case, if you add a separate column into your view, as defined here, you then need to add an appropriate mapping below that to show the new item.

<RelativeLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 android:layout\_width="match\_parent"
 android:layout\_height="60dp"
 android:padding="@dimen/standard\_medium\_margin">
 <LinearLayout
 android:layout\_width="match\_parent"
 android:layout\_height="match\_parent">
 <TextView
 android:id="@+id/dive\_number"
 android:layout\_width="50dp"
 android:layout\_height="wrap\_content"
 android:gravity="center\_vertical"/>
 <TextView
 android:id="@+id/dive\_name"
 android:layout\_width="334dp"
 android:layout\_height="46dp"
 android:gravity="center\_vertical"
 android:textColor="#FF0000"
 tools:layout\_editor\_absoluteX="42dp"
 tools:layout\_editor\_absoluteY="8dp"/>
 </LinearLayout>
</RelativeLayout>

You can fill out “dive\_number” text item with the following line added to your mappings

fragment.setMapping(R.id.*dive\_number*, new MappingData("number", MappingData.MappingType.*TEXT*));

# Editing a List Item

Editing a list item is accomplished using an Activity that we’ve designed to do most of the work for you. So, to begin with, you really need to do very little to create editing and creation functionality for your list. As you get more confident about things and want to take more control over how your application works, you can add more design and functionality to your app.

So, to being, you need to create a new activity. We’ve found that the best one to start is the standard Android “Scrolling Activity”. This is a good way to start as quite often there will be quite a few things to edit and so your edit activity will need to scroll.

When you’ve done this, you should find that this will have created an activity\_whatever\_you\_called\_it.xml file and a content\_whatever\_your\_called\_it.xml file. You need to into the content file and replace it with the following snippet

<android.support.v4.widget.NestedScrollView
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 xmlns:tools="http://schemas.android.com/tools"
 android:layout\_width="match\_parent"
 android:layout\_height="match\_parent"
 app:layout\_behavior="@string/appbar\_scrolling\_view\_behavior"
 tools:context=".activities.DiveEditActivity"
 tools:showIn="@layout/activity\_dive\_edit">

 <fragment
 android:id="@+id/dive\_edit\_fragment"
 android:name="io.linkalist.linkalist\_client.fragments.ItemEditFragment"
 android:layout\_width="match\_parent"
 android:layout\_height="match\_parent"
 tools:layout="@layout/fragment\_item\_edit"/>

</android.support.v4.widget.NestedScrollView>

As you can see, this replaces the standard scrolling content with an ItemEditFragment that does most of the heavy lifting for us.

## Coding the Editor

Now, we should go back into our WhateverYouCalledItActivity and set up the code properly. As we’ve implemented our generic editing activity as “ItemActivity” the quickest way to get started it is to make your activity just extent this activity.

public class DiveEditActivity extends ItemEditActivity
{
 public static void launchActivity(Activity parent, long listID, Item listItem)
 {
 ItemEditActivity.*launchActivity*(parent, DiveEditActivity.class, listID, listItem);
 }
}

Next, you need to create the launcher – this is a static function that you can use to launch the activity cleanly. You don’t need to do this - You can launch the activity on an intent with the list ID and item ID as arguments (LinkalistActivity.ARG\_LIST\_ID, linkalistActivity.ARG\_ITEM\_ID). But it is generally simple to create a launcher function as above.

Now, we are almost done except for a little housekeeping in the on Create function which should already have code in it to set the content view and set up the toolbar. You’ll need to leave these and add calls to set up the fragment ID and the send button ID to suit the underlying activity. Once you’ve done this, you should be good to go.

@Override
protected void onCreate(Bundle savedInstanceState)
{
 super.onCreate(savedInstanceState);

 setContentView(R.layout.*activity\_dive\_edit*);
 Toolbar toolbar = findViewById(R.id.*toolbar*);
 setSupportActionBar(toolbar);

 setEditFragmentID(R.id.*dive\_edit\_fragment*);

 setSendButtonID(R.id.*fab*);
 initialize();;
}

## Launching the Activity.

Now, we need to go back to your list activity and register this as the list Fragment's “Click Listener”. This is done by adding the highlighted line to the Activity’s detailsCompleted function. This will throw up an error saying your activity cannot be converted to a ListClick Listener. This can be fixed my letting you activity implement the ListClickListener interface.

@Override
public void detailsCompleted()
{
 FragmentManager fm = getSupportFragmentManager();
 SimpleListFragment fragment = (SimpleListFragment) fm.findFragmentById(R.id.*fragment\_instance*);
 if (null != fragment)
 {
 fragment.setListName("Your Dives");
 fragment.setDefaultCellLayout(R.layout.*list\_item\_dive*);
 fragment.setMapping(R.id.*dive\_number*, new MappingData("no", MappingData.MappingType.*TEXT*));
 fragment.setMapping(R.id.*dive\_name*, new MappingData("name", MappingData.MappingType.*TEXT*));
 fragment.setMapping(R.id.*dive\_max\_depth*, new MappingData("max\_depth", MappingData.MappingType.*TEXT*).setFormat("${value}m."));
 fragment.initialise();

 fragment.setClickListener(this);
 }
}

This will in turn prompt you to add a listItemClick function to your activity which you should do. This is where you’ll need to call your Edit class’s launcher function.

@Override
public void listItemClick(long listID, Item listItem)
{
 DiveEditActivity.*launchActivity*(this, listID, listItem);
}

Now, the application should have the two activities tied together and you should be ready to roll.



## Lists, List Views and Item Views

Now that you’ve got your basic list and editor up and running you’ll have possibly gotten a little confused by the treatment of lists and views. In linkalist, the underlying data storage object is a List. However, lists may only be accessed by attaching a view to the list. This view controls which columns get displayed, allows conditional display of specific items and allows control how the data is sorted.

So far, so good unless you start having multiple views on the same lists. This is all fine except that your application will now see two different versions of the same list. You’ll notice that when examining the API data, that two lists will have the same “model\_id”. This is general not a problem as your application can generally quite happy treat the two separate views on the same list as entirely separate entities.

The final piece of the puzzle is the item view. This allows you to design the order and various other display options for the fields in a list item completely independently of the view.

## Defining what you Edit

A list view allows you to define which columns appear in the list. Since the same list is used on the web version of you application as on the mobile app version, the list of columns that can be display in the list itself is often quite restricted. As such, you may define certain columns on the view as being available in the view as follows

* **Item Only**: These columns will only appear when looking at an item. These are used in advanced web views with a pop-up editor but will also appear in list data on the web application.
* **API Lists**: This allows hidden columns to be forced to appear in API queries. Essentially, this allows you to hide columns in the web version of your app.

In effect, pretty much all columns will be available to edit on your client application. The only exception are hidden columns that aren’t flagged as API Lists.

## Using the Item View

When you use the item view, this allows you to edit a completely different set of columns to that available in the list. In many cases, it will hurt performance to include all of the list item data in the list – for example if there is a large block of text. Hence, we can eliminate these heavy columns from the list query and only load them when we need to edit a single item. This is achieved by setting up an item view for the list, and then flagging it as “Use for Editing”.

When this is turned on, the ItemEditFragment behaves quite differently. When the fragment loads, it tries to display what data it has and then issues a request to the server for the details for a single item. When this returns, the fragment is filled out but the data you edit is stored separately to that of the list view version of the item. When you save the data, the post response will return both updated versions of the item and everything is kept in sync.

In practice, you don’t need to worry about this. Basically, the system takes care of it – the only real effect is a slight delay on displaying the data when editing. Once you have opened the item, any subsequent visits to the item should have the data all in place.

## Coding Options for Editing

There are three options for editing your list which give you varying degrees of control over how your edit activity looks

### Using ItemEditActivity

The linkalist ItemEditActivity library class will automatically configure itself based on your application’s configuration. All you need to do is pass it a list ID and an item to allow it to boot itself.

ItemEditActivity.*launchActivity*(this, listID, item);

This the lowest effort way of doing this and will give you the basic functionality that you need to edit an item.

### Using ItemEditFragment with your own Activity

The next stage of customisation is to construct your activity that extends ItemEditActivity. This will allow you some control over the way that your activity interacts with the edit panel but linkalist will still retain control over how the edit panel works. Your activity will need use an ItemEditFragment in its layout.

<fragment
 android:id="@+id/dive\_edit\_fragment"
 android:name="io.linkalist.linkalist\_client.fragments.ItemEditFragment"
 android:layout\_width="match\_parent"
 android:layout\_height="match\_parent"
 tools:layout="@layout/fragment\_item\_edit"/>

You’ll also need to add a few lines of code into the onCreate function for your activity.

@Override
protected void onCreate(Bundle savedInstanceState)
{
 super.onCreate(savedInstanceState);
 setContentView(R.layout.*activity\_dive\_edit*);
 setEditFragmentID(R.id.*dive\_edit\_fragment*);
 setSendButtonID(R.id.*fab*);
 initialize();
}

For simplicity you should also create your launcher functions but this is not mandatory.

public class DiveEditActivity extends ItemEditActivity
{
 public static void launchActivity(Activity parent, long listID, Item listItem)
 {
 ItemEditActivity.*launchActivity*(parent, DiveEditActivity.class, listID, listItem);
 }

 public static void launchActivityForNew(Activity parent, long listID)
 {
 ItemEditActivity.*launchActivityForNew*(parent, DiveEditActivity.class, listID);
 }

 ....

}

And that is about all you need to do. You can extend any of the built-in functions as required.

### Using your own ItemEditFragment

Building on the activity you created in the previous step, you can create your own extension of ItemEditFragment to gain more complete control over the behaviour. This will allow you to define exactly how your edit form is laid out and how the various data sources map to it.

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
 Bundle savedInstanceState)
{
 // Inflate the layout for this fragment
 return inflater.inflate(R.layout.*fragment\_dive\_edit*, container, false);
}

public void initialize()
{
 View rootView = getView();
 if (null != rootView)
 {
 if (getView() instanceof ViewGroup)
 {
 addMapping(rootView, R.id.*edit\_dive\_id*, "id");
 addMapping(rootView, R.id.*edit\_dive\_number*, "number");
 addMapping(rootView, R.id.*edit\_dive\_name*, "name");
 addMapping(rootView, R.id.*edit\_dive\_image*, "photo");
 addMapping(rootView, R.id.*edit\_dive\_surface\_interval*, "surface\_interval");
 addMapping(rootView, R.id.*edit\_dive\_date*, "date");
 addMapping(rootView, R.id.*edit\_dive\_weather\_conditions*, "weather\_conditions");
 addMapping(rootView, R.id.*edit\_dive\_site*, "site\_id");
 }
 }
}

Once you define these mappings the underlying code takes care of getting the data in and out of the structures onto the form. However, you must ensure that the correct type match is set up. For Text View controls, you may map these onto either an EditText or an EditText wrapped in a TextInputLayout. If using the latter, make sure that the layout itself gets the control ID used the in the addMapping function.

For all other data types, you may map onto the control type directly or use a linkalist wrapper type that also includes the control itself. If you get the wrong mapping, nothing too bad should happen – just the control won’t be displayed.

Note that you can’t map onto the id column.

These mappings are as follows

|  |  |  |
| --- | --- | --- |
| Data Type | Control Type | Wrapper Type  |
| Text (Fixed, Short or Long) | EditText | TextInputLayout |
| Number/Decimal | EditText | TextInputLayout |
| Currency | EditText | TextInputLayout |
| Date | DateEditView | TextInputLayout |
| Time | TimeEditView | TextInputLayout |
| Point | MapView / SquareMapView | MapViewLayout |
| Selection | Spinner | SpinnerInputLayout |
| Image | ImageSelectionView, ImageView | ImageInputLayout |
| Reference | Spinner | SpinnerInputLayout |
| DateTime, Timestamp | Not yet supported |  |
| Yes/No | Not yet supported |  |

## Sending Data

For the latter two methods, you need to do a little bit of work to send the data. For both cases you need to make sure that the ItemEditFragment is aware of the ID of the send button before you initialize it. This is illustrated in the code snippet for onCreate in the ItemEditActivity

 setSendButtonID(R.id.*fab*);

Alternatively, you can take care of this yourself by just calling sendData in your ItemEditFragment.