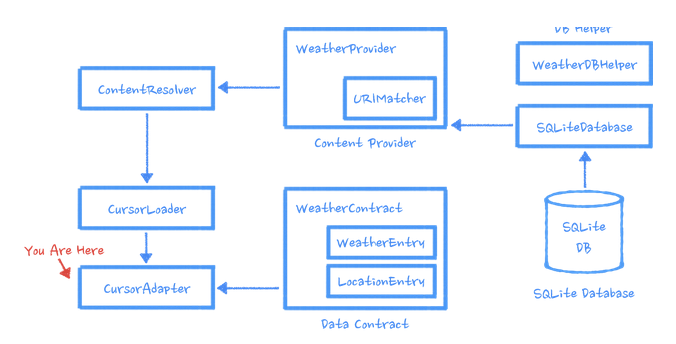
**Using CursorAdapter**

**The Plan**



As mentioned in the previous node, we’re going to be adding a CursorAdapter to Sunshine called ForecastAdapter. Why? Well, our loader will be making the calls on our content provider to get a Cursorand we’ll want to take the data from that cursor and put it into our UI. This is what adapters are meant for, associating data with UI components.

For now, we’re not going to worry about the Loader and we're just going to change our code to use a more appropriate adapter. Right now, you might recall from [**lesson 1**](https://classroom.udacity.com/courses/ud853/lessons/1395568821/concepts/16040297570923), we’re using an ArrayAdapter. This ArrayAdapter is populated only when we’re syncing with OpenWeatherMapAPI. Basically we get the JSON, put it in the content provider, take it out again, and change it to an array. This is not ideal. Let’s fix all of this.

We’ll talk more about making an adapter in Lesson 5, for now, we’re going to give you the code for one and do a quick overview.

Go ahead and copy **[ForecastAdapter](https://github.com/udacity/Sunshine-Version-2/blob/4.18_cursor_adapter/app/src/main/java/com/example/android/sunshine/app/ForecastAdapter.java" \t "_blank)** and [**Utility.java**](https://github.com/udacity/Sunshine-Version-2/blob/4.18_cursor_adapter/app/src/main/java/com/example/android/sunshine/app/Utility.java) into the main package (example.android.com) of your code.

Also add [WeatherProvider](https://gist.github.com/teohaik/3fcd2c76e6cb78a5ac30ef9485220ea8) class and modify the manifest.xml accordingly

|  |
| --- |
| …  <**provider  android:authorities="com.example.android.sunshine.app"  android:name=".data.WeatherProvider"  android:enabled="true"  android:exported="true"** >  </**provider**>  </**application**> </**manifest**> |

**What’s going inside ForecastAdapter**

ForecastAdapter is a subclass of CursorAdapter. Let’s take a quick look at the functionality we’ve got in here. After the constructor, there are four methods. The first two are formatHighLows and convertCursorRowToUXFormat. These are two formatting methods specific to Sunshine.

* convertCursorRowToUXFormat takes a row from a cursor and constructs a single string of the format:

Date - Weather -- High/Low

This is the string we’re used to seeing in the listview element. It uses formatHighLow to get the correct string for the temperature.

The other two methods are necessary to override whenever you’re extending a cursor adapter.

* newView - Remember that adapters work with listviews to populate them. They create duplicates of the same layout to put into the list view. This is where you return what layout is going to be duplicated.
* View view = LayoutInflater.from(context).inflate(R.layout.list\_item\_forecast, parent, false);
* return view;

In our case, we’re inflating our listview layout, list\_item\_forecast, and then returning it.

* bindView - This is where the exciting bit occurs. As the name suggests you are **binding** the values in the cursor to the view.
* TextView tv = (TextView)view;
* tv.setText(convertCursorRowToUXFormat(cursor));

The View passed into bindView is the View returned from newView. We know it’s a TextView, so we cast it. Then we take the Cursor, run it through our custom made formatting function, and set the text of the TextView.

**Refactoring to use ForecastAdapter with Cursors and from the Fragment**

**1. Change mForecastAdapter's type**

Change mForecastAdapter, to be an instance of ForecastAdapter.

**2. Get Data from the Database**

Let’s go to where we first need to populate the ForecastFragment with data and do so by getting the data from the database. Go to onCreateView. Use WeatherProvider to query the database the same way you are in FetchWeatherTask:

String locationSetting = Utility.getPreferredLocation(getActivity());

// Sort order: Ascending, by date.

String sortOrder = WeatherContract.WeatherEntry.COLUMN\_DATE + " ASC";

Uri weatherForLocationUri = WeatherContract.WeatherEntry.buildWeatherLocationWithStartDate(

locationSetting, System.currentTimeMillis());

Cursor cur = getActivity().getContentResolver().query(weatherForLocationUri,

null, null, null, sortOrder);

**3. Make a new ForecastAdapter**

Still in onCreateView, we have a Cursor cur, so let’s use our new ForecastAdapter. Create a new ForecastAdapter with the new cursor. The list will be empty the first time we run.

mForecastAdapter = new ForecastAdapter(getActivity(), cur, 0);

**4. Delete OnItemClickListener**

Because we changed the adapter, the OnItemClickListener in ForecastFragment for the ListView won’t work. Specifically this line String forecast = mForecastAdapter.getItem(position); is problematic because getItem with a CursorAdapter doesn’t return a string.

Go ahead and remove or comment out this for now.

We’ll talk more about this and correct this soon enough. Until then, our code will compile and run but not have access to our DetailView.

**5. Clean up**

Inside of FetchWeatherTask, we’re going to remove the formatting code and anything for updating the adapter. You can remove:

* Any reference to mForecastAdapter
* getReadableDateString, formatHighLows, convertContentValuesToUXFormat. These are all formatting functions and we’ve moved them to the ForecastAdapter.
* The lines in getWeatherDataFromJson where we requery the database after the insert.
* PostExecute

**Note:** To keep your tests working, you'll need to modify line 42 of TestFetchWeatherTask to be:

FetchWeatherTask fwt = new FetchWeatherTask(getContext());

Use of Loaders in Forecast Fragment

public class ForecastFragment extends Fragment implements LoaderManager.LoaderCallbacks<Cursor> {

private static final int FORECAST\_LOADER = 0;

private ForecastAdapter mForecastAdapter;

@Override

public void onActivityCreated(Bundle savedInstanceState) {

getLoaderManager().initLoader(FORECAST\_LOADER, null, this);

super.onActivityCreated(savedInstanceState);

}

private void updateWeather() {

FetchWeatherTask weatherTask = new FetchWeatherTask(getActivity());

String location = Utility.getPreferredLocation(getActivity());

weatherTask.execute(location);

}

@Override

public void onStart() {

super.onStart();

updateWeather();

}

@Override

public Loader<Cursor> onCreateLoader(int i, Bundle bundle) {

String locationSetting = Utility.getPreferredLocation(getActivity());

// Sort order: Ascending, by date.

String sortOrder = WeatherContract.WeatherEntry.COLUMN\_DATE + " ASC";

Uri weatherForLocationUri = WeatherContract.WeatherEntry.buildWeatherLocationWithStartDate(

locationSetting, System.currentTimeMillis());

return new CursorLoader(getActivity(),

weatherForLocationUri, null, null, null, sortOrder);

}

@Override

public void onLoadFinished(Loader<Cursor> cursorLoader, Cursor cursor) {

mForecastAdapter.swapCursor(cursor);

}

@Override

public void onLoaderReset(Loader<Cursor> cursorLoader) {

mForecastAdapter.swapCursor(null);

}

**Projections**

private static final String[] FORECAST\_COLUMNS = {

// In this case the id needs to be fully qualified with a table name, since

// the content provider joins the location & weather tables in the background

// (both have an \_id column)

// On the one hand, that's annoying. On the other, you can search the weather table

// using the location set by the user, which is only in the Location table.

// So the convenience is worth it.

WeatherContract.WeatherEntry.TABLE\_NAME + "." + WeatherContract.WeatherEntry.\_ID,

WeatherContract.WeatherEntry.COLUMN\_DATE,

WeatherContract.WeatherEntry.COLUMN\_SHORT\_DESC,

WeatherContract.WeatherEntry.COLUMN\_MAX\_TEMP,

WeatherContract.WeatherEntry.COLUMN\_MIN\_TEMP,

WeatherContract.LocationEntry.COLUMN\_LOCATION\_SETTING,

WeatherContract.WeatherEntry.COLUMN\_WEATHER\_ID,

WeatherContract.LocationEntry.COLUMN\_COORD\_LAT,

WeatherContract.LocationEntry.COLUMN\_COORD\_LONG

};

// These indices are tied to FORECAST\_COLUMNS. If FORECAST\_COLUMNS changes, these

// must change.

static final int COL\_WEATHER\_ID = 0;

static final int COL\_WEATHER\_DATE = 1;

static final int COL\_WEATHER\_DESC = 2;

static final int COL\_WEATHER\_MAX\_TEMP = 3;

static final int COL\_WEATHER\_MIN\_TEMP = 4;

static final int COL\_LOCATION\_SETTING = 5;

static final int COL\_WEATHER\_CONDITION\_ID = 6;

static final int COL\_COORD\_LAT = 7;

static final int COL\_COORD\_LONG = 8;

private String convertCursorRowToUXFormat(Cursor cursor) {

String highAndLow = formatHighLows(

cursor.getDouble(ForecastFragment.COL\_WEATHER\_MAX\_TEMP),

cursor.getDouble(ForecastFragment.COL\_WEATHER\_MIN\_TEMP));

return Utility.formatDate(cursor.getLong(ForecastFragment.COL\_WEATHER\_DATE)) +

" - " + cursor.getString(ForecastFragment.COL\_WEATHER\_DESC) +

" - " + highAndLow;

}

**Make Details View Functional**

One of the things that we decided to temporarily break is the details view. It’s time to fix this and hook things up.

The major change we will make here is that we will start our DetailsActivity by passing it the URI it needs to pass to the content provider to get the correct data.

**1. Add OnItemClickListener to ListView**

In ForecastFragment, in the onCreateView method, go ahead and add an onItemClickListener, except this time, it’s going to pass a URI for the data needed for the detail view.

listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {

@Override

public void onItemClick(AdapterView adapterView, View view, int position, long l) {

// CursorAdapter returns a cursor at the correct position for getItem(), or null

// if it cannot seek to that position.

Cursor cursor = (Cursor) adapterView.getItemAtPosition(position);

if (cursor != null) {

String locationSetting = Utility.getPreferredLocation(getActivity());

Intent intent = new Intent(getActivity(), DetailActivity.class)

.setData(WeatherContract.WeatherEntry.buildWeatherLocationWithDate(

locationSetting, cursor.getLong(COL\_WEATHER\_DATE)

));

startActivity(intent);

}

}

});

**2. Print the URI in the ListView**

On the DetailActivity side, we'll want to change the code, which is referring to an intent extra that you're no longer setting. Instead we used setData so we need to grab this data using getDataString. The full code you'll need to put in DetailActivity is:

if (intent != null) {

mForecastStr = intent.getDataString();

}

This causes the detail view to show the URI.

But wait, that's not what we want! So obviously we’re **not** done at this point, we need to actually use the URI to display the correct data in the detail view. You’ll be doing that in the next node.

Check out the full [**diff here**](https://github.com/udacity/Sunshine-Version-2/compare/4.20_projections...4.21_details_view).

**Implement Details View as Cursor Loader (+use Projections)**

|  |
| --- |
| public static class DetailFragment extends Fragment implements LoaderCallbacks<Cursor> { |
| private static final String LOG\_TAG = DetailFragment.class.getSimpleName(); |
| private static final String FORECAST\_SHARE\_HASHTAG = " #SunshineApp"; |
| private ShareActionProvider mShareActionProvider; |
| private String mForecast; |
| private static final int DETAIL\_LOADER = 0; |
|  |
| private static final String[] FORECAST\_COLUMNS = { |
| WeatherEntry.TABLE\_NAME + "." + WeatherEntry.\_ID, |
| WeatherEntry.COLUMN\_DATE, |
| WeatherEntry.COLUMN\_SHORT\_DESC, |
| WeatherEntry.COLUMN\_MAX\_TEMP, |
| WeatherEntry.COLUMN\_MIN\_TEMP, |
| }; |
| // these constants correspond to the projection defined above, and must change if the |
| // projection changes |
| private static final int COL\_WEATHER\_ID = 0; |
| private static final int COL\_WEATHER\_DATE = 1; |
| private static final int COL\_WEATHER\_DESC = 2; |
| private static final int COL\_WEATHER\_MAX\_TEMP = 3; |
| private static final int COL\_WEATHER\_MIN\_TEMP = 4; |
|  |
| public DetailFragment() { |
| setHasOptionsMenu(true); |
| } |
| @Override |
| public View onCreateView(LayoutInflater inflater, ViewGroup container, |
| Bundle savedInstanceState) { |
| return inflater.inflate(R.layout.fragment\_detail, container, false); |
| } |
| @Override |
| public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) { |
| // Inflate the menu; this adds items to the action bar if it is present. |
| inflater.inflate(R.menu.detailfragment, menu); |
| // Retrieve the share menu item |
| MenuItem menuItem = menu.findItem(R.id.action\_share); |
|  |
| // Get the provider and hold onto it to set/change the share intent. |
| mShareActionProvider = (ShareActionProvider) MenuItemCompat.getActionProvider(menuItem); |
| // If onLoadFinished happens before this, we can go ahead and set the share intent now. |
| if (mForecast != null) { |
| mShareActionProvider.setShareIntent(createShareForecastIntent()); |
| } |
| } |
| private Intent createShareForecastIntent() { |
| Intent shareIntent = new Intent(Intent.ACTION\_SEND); |
| shareIntent.addFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_WHEN\_TASK\_RESET); |
| shareIntent.setType("text/plain"); |
| shareIntent.putExtra(Intent.EXTRA\_TEXT, mForecast + FORECAST\_SHARE\_HASHTAG); |
| return shareIntent; |
| } |
| @Override |
| public void onActivityCreated(Bundle savedInstanceState) { |
| getLoaderManager().initLoader(DETAIL\_LOADER, null, this); |
| super.onActivityCreated(savedInstanceState); |
| } |
| @Override |
| public Loader<Cursor> onCreateLoader(int id, Bundle args) { |
| Log.v(LOG\_TAG, "In onCreateLoader"); |
| Intent intent = getActivity().getIntent(); |
| if (intent == null) { |
| return null; |
| } |
| // Now create and return a CursorLoader that will take care of |
| // creating a Cursor for the data being displayed. |
| return new CursorLoader( |
| getActivity(), |
| intent.getData(), |
| FORECAST\_COLUMNS, null, null, null); |
|  |
| ); |
| } |
| @Override |
| public void onLoadFinished(Loader<Cursor> loader, Cursor data) { |
| Log.v(LOG\_TAG, "In onLoadFinished"); |
| if (!data.moveToFirst()) { return; } |
| String dateString = Utility.formatDate( |
| data.getLong(COL\_WEATHER\_DATE)); |
| String weatherDescription = |
| data.getString(COL\_WEATHER\_DESC); |
| boolean isMetric = Utility.isMetric(getActivity()); |
| String high = Utility.formatTemperature( |
| data.getDouble(COL\_WEATHER\_MAX\_TEMP), isMetric); |
| String low = Utility.formatTemperature( |
| data.getDouble(COL\_WEATHER\_MIN\_TEMP), isMetric); |
| mForecast = String.format("%s - %s - %s/%s", dateString, weatherDescription, high, low); |
| TextView detailTextView = (TextView)getView().findViewById(R.id.detail\_text); |
| detailTextView.setText(mForecast); |
| // If onCreateOptionsMenu has already happened, we need to update the share intent now. |
| if (mShareActionProvider != null) { |
| mShareActionProvider.setShareIntent(createShareForecastIntent()); |
| } |
| } |
| @Override |
| public void onLoaderReset(Loader<Cursor> loader) { } |
| } |
|  |

## Handle the Setting Change

One more thing. Right now we're no longer responding correctly when a setting is changed. For example, if you were to update the location setting from Mountain View California, to say, Rochester New York, Sunshine won't work. Why? Well, it would look for the data in the database, but you don't have that data for a new location. Let's fix this.

**1. Create mLocation**

In MainActivity create a mLocation variable to store our current known location.

Initialize this variable in onCreate to be whatever is currently stored in settings.

**2. Add a Fragment Tag**

Add a constant for a fragment tag called FORECASTFRAGMENT\_TAG. A fragment tag is a constant String we can use to tag a fragment within the fragment manager so we can easily look it up later.

When we add the fragment, we add the parameter for the FORECASTFRAGMENT\_TAG. Fragments can only be tagged during the fragment transaction. This means that the same fragment code can be reused multiple times with different tags.

if (savedInstanceState == null) {

getSupportFragmentManager().beginTransaction()

.add(R.id.container, new ForecastFragment(), FORECASTFRAGMENT\_TAG)

.commit();

}

**3. Create onLocationChanged method**

In our ForecastFragment, create onLocationChanged. This method will do two things, it will first call updateWeather and then it will restart the loader.

**4. Check for Location Change**

Now in onResume in MainActivity, check whether the location has changed by comparing whatever is stored in the settings (ie Utility.getPreferredLocation(this);) with mLocation. If it has changed:

1. Get the ForecastFragment using the tag FORECASTFRAGMENT\_TAG by using:

ForecastFragment ff =

(ForecastFragment)getSupportFragmentManager().findFragmentByTag(FORECASTFRAGMENT\_TAG);

1. Call onLocationChanged
2. Update mLocation

**5. Remove Excessive Weather Fetching**

Now that we have a database, we don’t have to constantly talk to the network and fetch the weather. But if you look at onStart from ForecastFragment, you'll see every time it's called, it downloads data from Open Weather Map. This means every time you rotate the device, you'll be attempting to connect to Open Weather Map. In Lesson 6 we’ll show you how to schedule updates in the background, but for now let’s save on network bandwidth and battery by **deleting onStart**. You can use the “refresh” menu item to get new weather data.