2021

EAD2 CA2 – Android Project

AN ALBUM TRACKER APP (ALBUMTRACKR)

RYAN DEERING X00144631 & JAMES LYNAM X00073019

TUD TALLAGHT CAMPUS | Main St, Tallaght

Table of Contents (Page Numbers will change)

Introduction

For our second CA in Enterprise Applications Development 2, we decided to create an app that stores lists of an individual's favourite albums into something like a playlist. The user will be able to create a list by assigning it a title and description, and once the list has been created, they would be able to add their favourite albums to this list.

When being prompted for input on the album, the Last.fm API would be queried, and if the album was real and present in their system, it would be added to the database. The user will also have the ability to scroll through different tabs based on different circumstances. These tabs would allow them to see their favourite album lists (through the use of a star method), the most popular album lists and the newest album lists.

There will also be some validation that would ensure that users can only delete and change their own lists, and we hope to accomplish this through the use of the device ID. The Web API methods will be created in ASP.NET Core and the client side of the application will be using Java in Android Studio.

We both have had a long history of interest in music, and believe that this interest is making this CA more engaging.

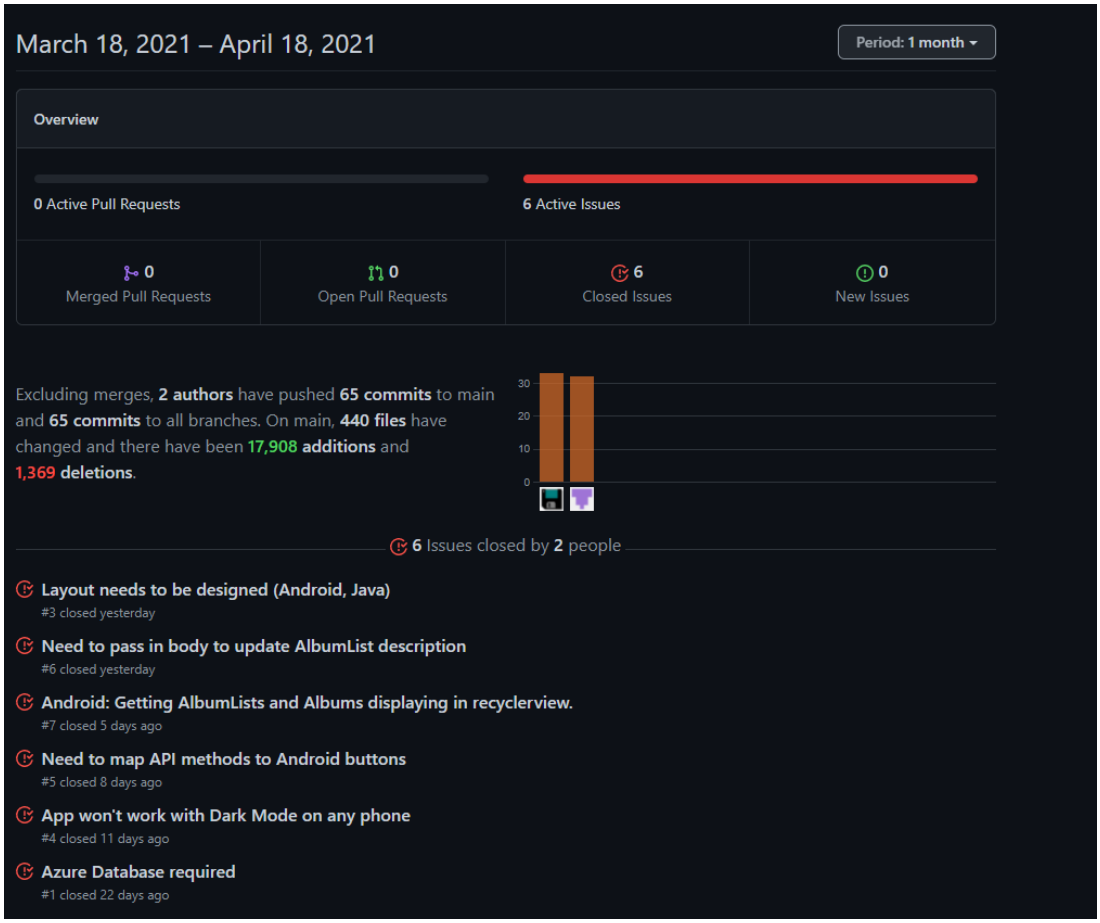
GitHub Information

Repo URI: https://github.com/ryandeering/albumtrackr.git

We have both made thorough use of the Github repository, and every time either of us made a commit, it automatically rebuilt the application in the pipeline on Azure. We tried to use a strict naming system with our commits in order to easier keep track of what was done. We did this by prefixing any commit with either “API” or “Android”, or in some cases, both. Github has been an amazing tool in this collaborative process, and we would have faced great difficulties without it.

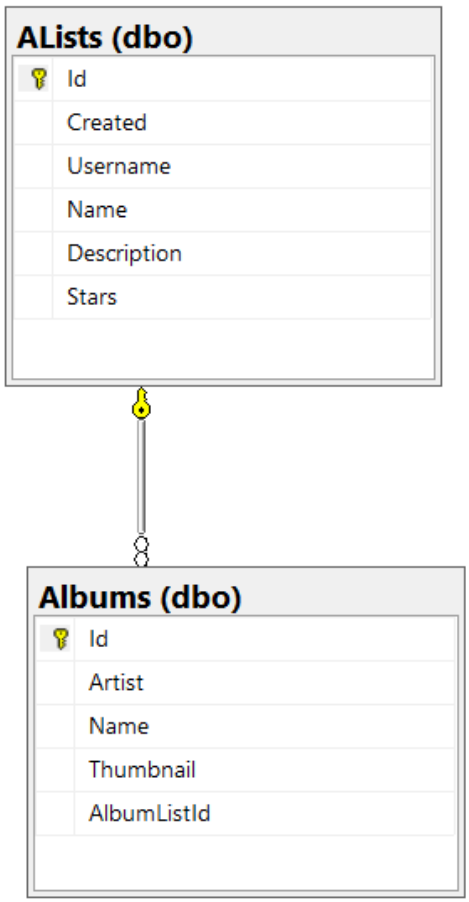
Commits:

Our most intensive week was the week before the CA was due. As a team, we often discussed what we were working on and any changes made were always confirmed with the other person. We also often found ourselves fixing small things that may not be significant, would be very useful in future.



Database Schema

Entity Relationship graph generated from SQL Server.



Average JSON response.

```
{
    "id": 23,
    "albums": [
        {
            "id": 26,
            "artist": "Ben Harper",
            "name": "Diamonds On The Inside",
            "thumbnail": "https://lastfm.freetls.fastly.net/i/u/300x300/742a8e58cef4207e286250a94ee9699b.png"
        },
        {
            "id": 27,
            "artist": "Jack Johnson",
            "name": "Sleep Through The Static",
            "thumbnail": "https://lastfm.freetls.fastly.net/i/u/300x300/ae444cac53ea4df1abb6a47311fc13be.png"
        }
    ],
    "created": "2021-04-18T05:43:17.6147932",
    "username": "6luserdebug3",
    "name": "James's Chill List",
    "description": "Music to relax to",
    "stars": 8
}
```


Service

Our Operations

GET Methods:

GetLatestLists(): This GET method returns the **latest** album lists that were added to the service. It does this by ordering the list in a descending order through the date of creation using LINQ.

```
/api/AlbumList
```

```
[HttpGet]
0 references
public async Task<IActionResult> GetLatestLists()
{
    return Ok(await _albumListRepository.GetLatestLists());
}
```

```
public async Task<List<AlbumList>> GetLatestLists()
{
    return await _albumtrackrContext.ALists.OrderByDescending(al :AlbumList => al.Created).Take(25).ToListAsync();
}
```

GetPopularLists(): This GET method returns the most **popular** lists in the service. It does this by ordering the list by the *count* of the stars that the list has received. For example, the list with the highest stars would be at the top of the lists using this method.

```
/api/AlbumList/popular/
```

```
[HttpGet("popular/")]
0 references
public async Task<IActionResult> GetPopularLists()
{
    return Ok(await _albumListRepository.GetPopularLists());
}
```

```
public async Task<List<AlbumList>> GetPopularLists()
{
    return await _albumtrackrContext.ALists.OrderByDescending(al :AlbumList => al.Stars).Take(25).ToListAsync();
}
```

GetMyLists(): This GET method returns the lists that have been added by the user with a device-specific username generated using from metadata of the device. This ensures that they can keep their own lists in a convenient location, and also helps identify which lists they can and cannot perform CRUD actions on. It does this by sorting the lists by ensuring that the username of the current device matches that of the created list. The username attribute is present in the **AlbumList** class.

```
/api/AlbumList/username/{username}
```

```
[HttpGet("username/{userName}")]
0 references
public async Task<IActionResult> GetMyLists(string userName)
{
    if (userName == null) return BadRequest();

    var userLists = await _albumListRepository.GetMyLists(userName);

    if (userLists == null) return NoContent();

    return Ok(userLists);
}
```

```
public async Task<List<AlbumList>> GetMyLists(string userName)
{
    return await _albumtrackrContext.ALists.Where(al :AlbumList => al.Username == userName).ToListAsync();
}
```

GetById(): This GET method the returns the lists for the **ID** that is supplied by the user. It does this by taking in the ID from the user, and comparing it to the ID present in the lists. It returns the first element if found, and throws a 404 if there is not a match.

```
/api/AlbumList/{id}
```

```
[HttpGet("{id}")]
0 references
public async Task<IActionResult> GetById(int id)
{
    var userList = await _albumListRepository.GetById(id);

    if (userList == null) return NotFound();

    return Ok(userList);
}
```

```
public async Task<AlbumList> GetById(int id)
{
    return await _albumtrackrContext.ALists.Include("Albums").FirstOrDefaultAsync(al AlbumList => al.Id == id);
}
```

POST Methods:

CreateAlbumList(): This POST method allows the user to create an AlbumList. It does this using the username, the name of the list and a description. Only the name of the list and the description need to be supplied by the user for the method to work. It works by essentially creating an AlbumList object based on these parameters.

```
/api/AlbumList
[HttpPost]
0 references
public async Task<IActionResult> CreateAlbumList([FromBody] AlbumList albumList)
{
    var list = await _albumListRepository.CreateAlbumList(albumList.Username, albumList.Name,
        albumList.Description);

    if (list == null) return BadRequest();

    return Ok(list);
}
```

```
public async Task<AlbumList> CreateAlbumList(string userName, string name, string description)
{
    var list = new AlbumList
    {
        Username = userName,
        Name = name,
        Description = description,
        Albums = new List<Album>(),
        Created = DateTime.Now,
        Stars = 0
    };

    await _albumtrackrContext.ALists.AddAsync(list);
    await _albumtrackrContext.SaveChangesAsync();


    return list;
}
```

AddToList(): This POST method allows the user to add an album to an existing AlbumList. The name of artist and album title are then used in a query to the Last.FM API, using the excellent Inflatable.Lastfm library for .NET Core. If the Album is null (in the case of non-existence, for example), a bad request is returned. Additionally, if the Album is found successfully, it will retrieve the album art for that album automatically.

```
/api/AlbumList/{id}/album
[HttpPost("{id}/album/")]
0 references
public async Task<IActionResult> AddToList(int id, [FromBody] Album album)
{
    if (album == null) return BadRequest();

    if (album.Artist == string.Empty || album.Name == string.Empty)
        ModelState.AddModelError("Artist/Album Name", "The artist or album name is not correct.");

    if (!ModelState.IsValid) return BadRequest(ModelState);

    var userList = await _albumListRepository.AddToList(id, album);

    return Ok(userList);
}
```

```
public async Task<AlbumList> AddToList(int id, Album album)
{
    var list = await _albumtrackrContext.ALists.Include("Albums").FirstAsync(al :AlbumList => al.Id == id);

    if (list == null) return null;

    list.Albums ??= new List<Album>();

    var apiKey :string = _configuration.GetSection( key: "LastFMApiKey").Value;
    var apiSecret :string = _configuration.GetSection( key: "LastFMApiSecret").Key;

    var client = new LastfmClient(apiKey, apiSecret);

    var response = await client.Album.GetInfoAsync( artistname: album.Artist,  albumname: album.Name);

    if (response.Content.Name != null || response.Content.ArtistName != null)
    {
        album.Thumbnail = response.Content.Images.Largest.ToString();
        album.Name = response.Content.Name;
        album.Artist = response.Content.ArtistName;
    }
    else
    {
        return null;
    }

    list.Albums.Add(album);
    await _albumtrackrContext.SaveChangesAsync();

    return list;
}
```

PUT Methods:

StarAlbumList(): This PUT method allows the user to “star” an AlbumList, this metric is used to determine the popularity of a list. It is a simple increment of that list. If the username of the user is matched to be the same as the list owner in the app, they can not increment the amount of stars.

```
/api/AlbumList/{id}/userName
```

```
// star an album by id
[HttpPut( template: "{albumListId}")]
public async Task<IActionResult> StarAlbumList(int albumListId)
{
    var userList = await _albumListRepository.StarAlbumList(albumListId);

    if (userList == null) return NotFound();

    return Ok(userList);
}
```

```
public async Task<AlbumList> StarAlbumList(int albumListId)
{
    var list = await _albumtrackrContext.ALists.Include("Albums") // IQueryable<AlbumList>
        .FirstOrDefaultAsync(al :AlbumList => al.Id == albumListId); // Task<AlbumList>

    if (list == null) return null;

    list.Stars += 1;
    _albumtrackrContext.ALists.Update(list);
    await _albumtrackrContext.SaveChangesAsync();
    return list;
}
```

EditDescription(): This PUT method allows the user to update the description of the AlbumList they are currently viewing. It does this by taking in the ID of the AlbumList, and updating the description of the user input to the Description attribute of the AlbumList class.

```
/api/AlbumList/{id}
[HttpPut("{id}")]
0 references
public async Task<IActionResult> EditDescription(int id)
{

    var userList = await _albumListRepository.EditDescription(id);

    return Ok(userList);
}
```

DELETE Methods:

DeleteList(): This DELETE method simply allows the user to delete an AlbumList. It used to ID to do this. If the ID is found, the list is deleted.

```
/api/AlbumList/{id}
[HttpDelete("{id}")]
0 references
public async Task<IActionResult> DeleteList(int id)
{
    if (id == null) return BadRequest();

    var userList = await _albumListRepository.DeleteList(id);

    return Ok(userList);
}
```

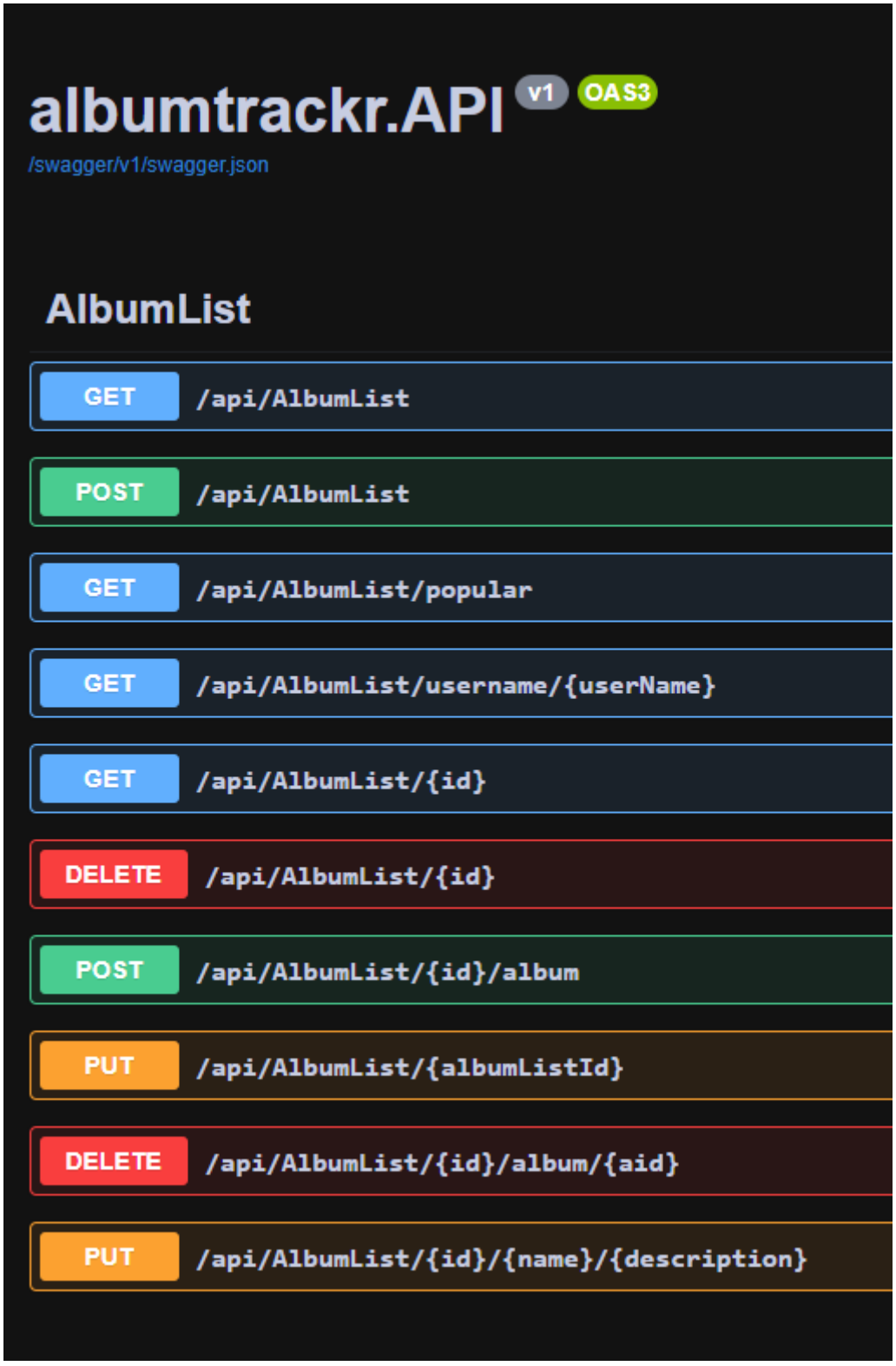
```
// Delete list
public async Task<AlbumList> DeleteList(int id)
{
    var list = await _albumtrackrContext.ALists.Include("Albums").FirstOrDefaultAsync(al :AlbumList => al.Id == id);

    if (list == null) return null;

    _albumtrackrContext.ALists.Remove(list);
    await _albumtrackrContext.SaveChangesAsync();

    return list;
}
```

DeleteFromList(): This DELETE method allows the user to delete an Album from an AlbumList. It uses the list ID and the Album ID to accomplish this. If the IDs match up, the album is removed from the list.

```
/api/AlbumList/{id}/album/{aid}
[HttpDelete("{id}/album/{aid}")]
0 references
public async Task<IActionResult> DeleteFromList(int id, int aid)
{
    if (aid == null) return BadRequest();

    var userList = await _albumListRepository.DeleteFromList(id, aid);

    return Ok(userList);
}
```

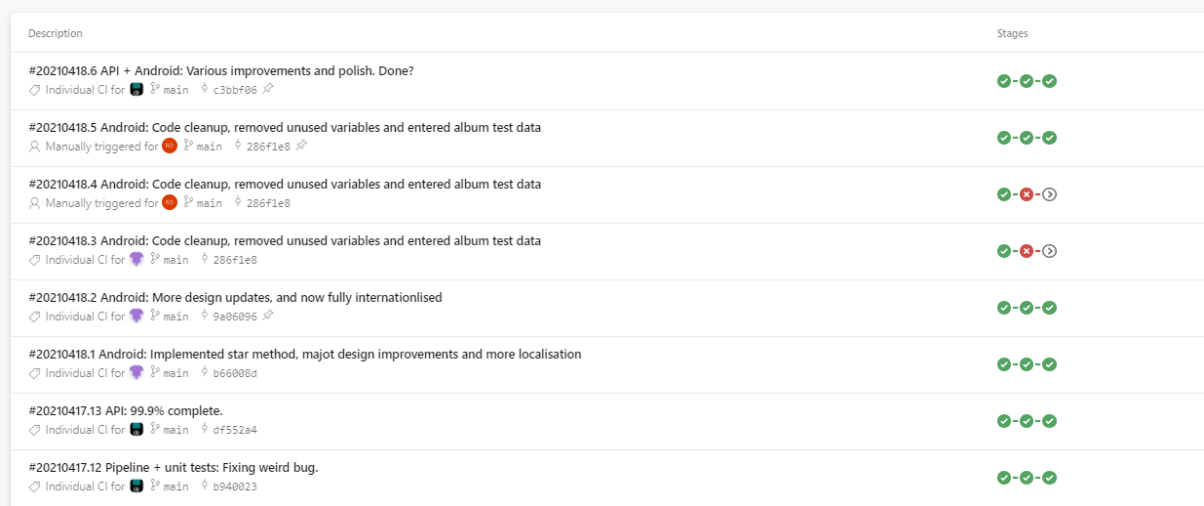
```
// Delete from list, album seems to still be present
public async Task<AlbumList> DeleteFromList(int id, int aid)
{
    var list = await _albumtrackrContext.ALists.Include("Albums").FirstOrDefaultAsync(al :AlbumList => al.Id == id);

    var album = list.Albums.First(a :Album => a.Id == aid);

    if (album == null) return null;

    list.Albums.Remove(album);
    await _albumtrackrContext.SaveChangesAsync();

    return list;
}
```

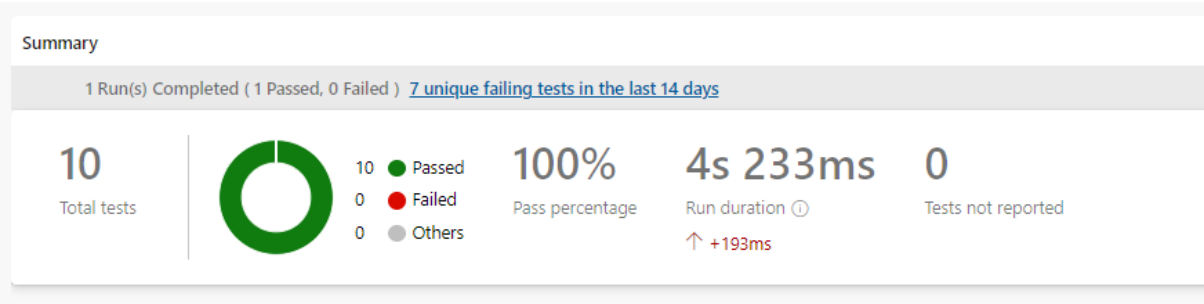
Screenshot of Swagger UI Test Page

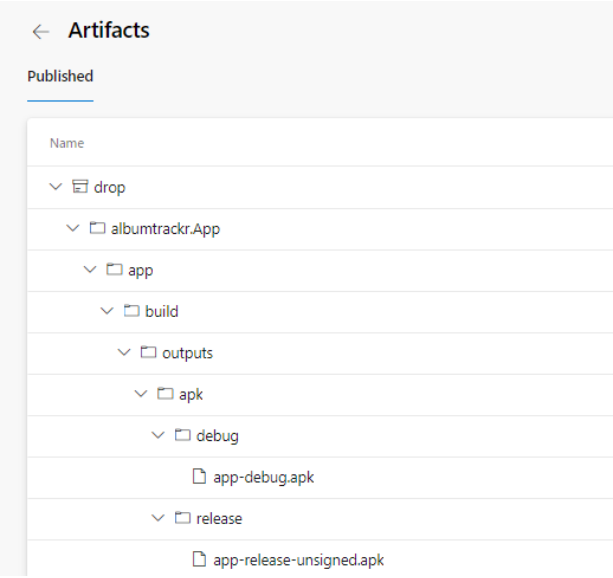


Deployment of Application

The application is deployed through a pipeline on Azure DevOps. Every time we make a commit to our Github repository, the .NET component of the device rebuilds and deploys itself onto a student free-tier Azure App Service hosting the API. The pipeline is ran through Azure DevOps, building and restoring the API first, then using Gradle to build the APK to publish an artifact consisting of a .ZIP file containing two APKs, a debug APK and a release APK for easy testing. We then have a testing stage in the pipeline, to run our XUnit unit tests to see if the logic of the device is working as it should.







Azure App Service Settings

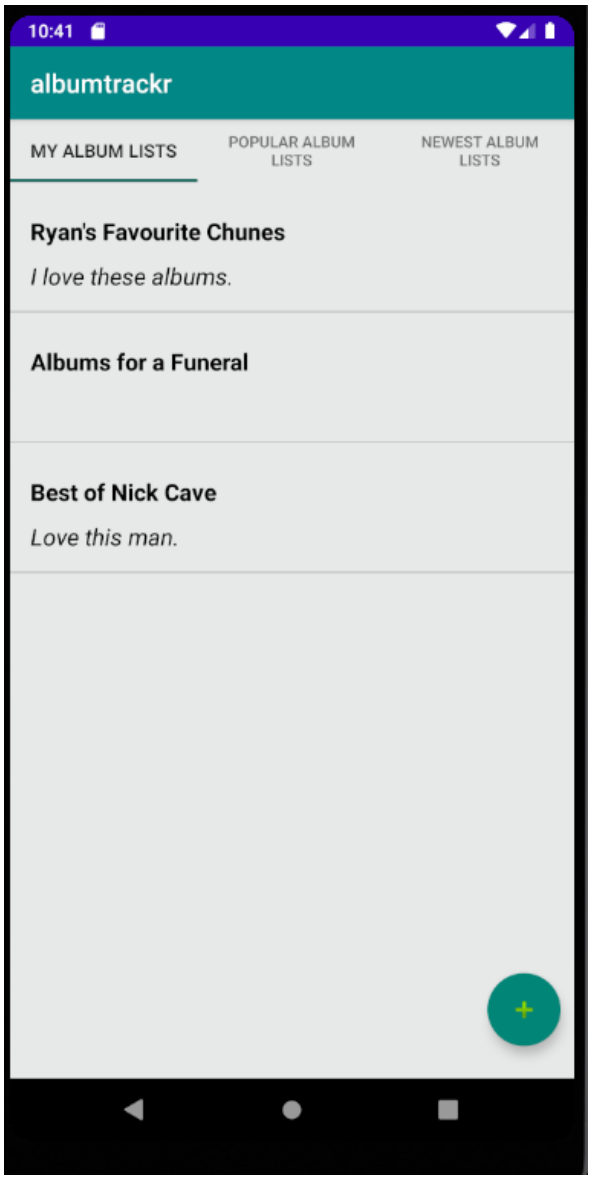
The application is deployed through a pipeline on Azure. Every time we make a commit to our GitHub repository, the pipeline is triggered to deploy the API onto it in a release configuration using the web deploy method. This avoids any read-only problems using run from package or run from zip often brings. Configuration of the App Service was not difficult, aside from getting around the default firewall settings and establishing our Azure databases connection string within the key store of the App Service. CORS was disabled, this is an unauthenticated API so there would be little point in configuring it. We used the logstream of the App Service often to diagnose potential problems with the deployment.

Application Breakdown

This section will contain screenshots of the applications features in use, a testing report, the process of internationalisation etc

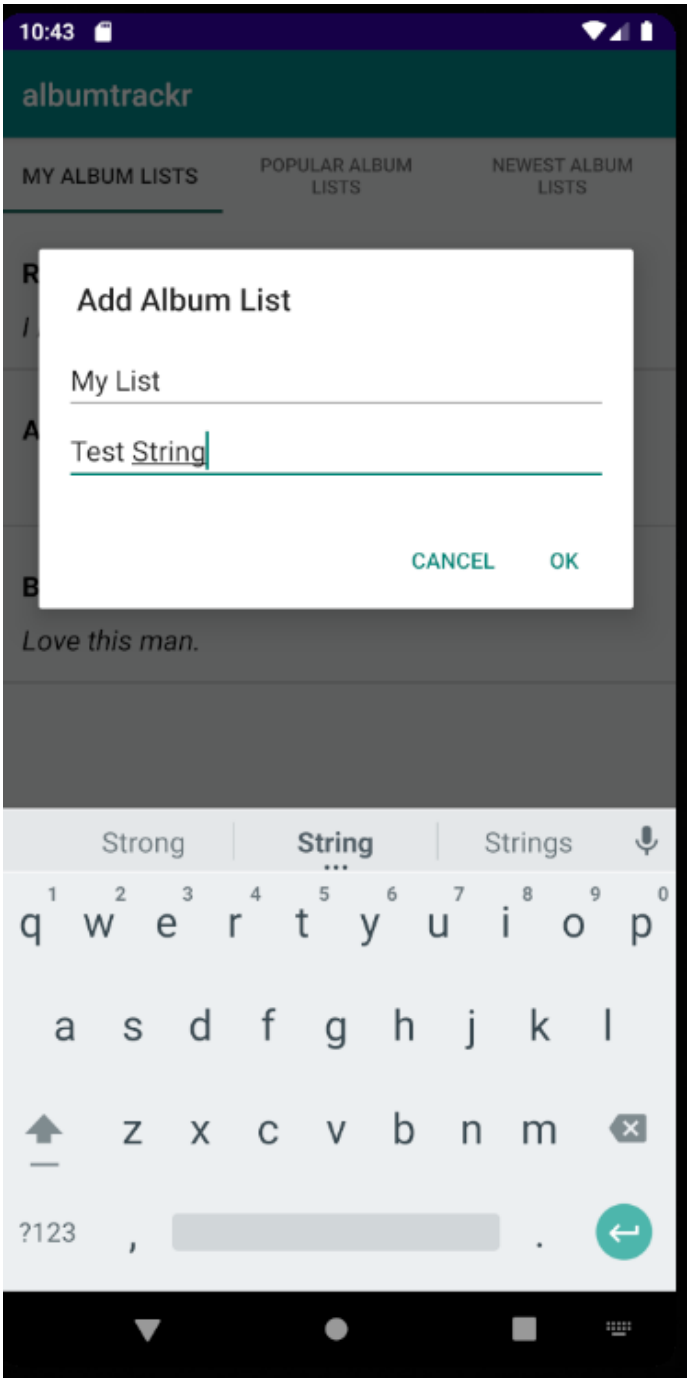
We have three fragments: **My Album Lists**, which contain lists with the same username of lists on the DB, **Popular Album Lists**, which contain the most starred lists on the DB and **Newest Album Lists**, which were the last ones to be created.

Fragments

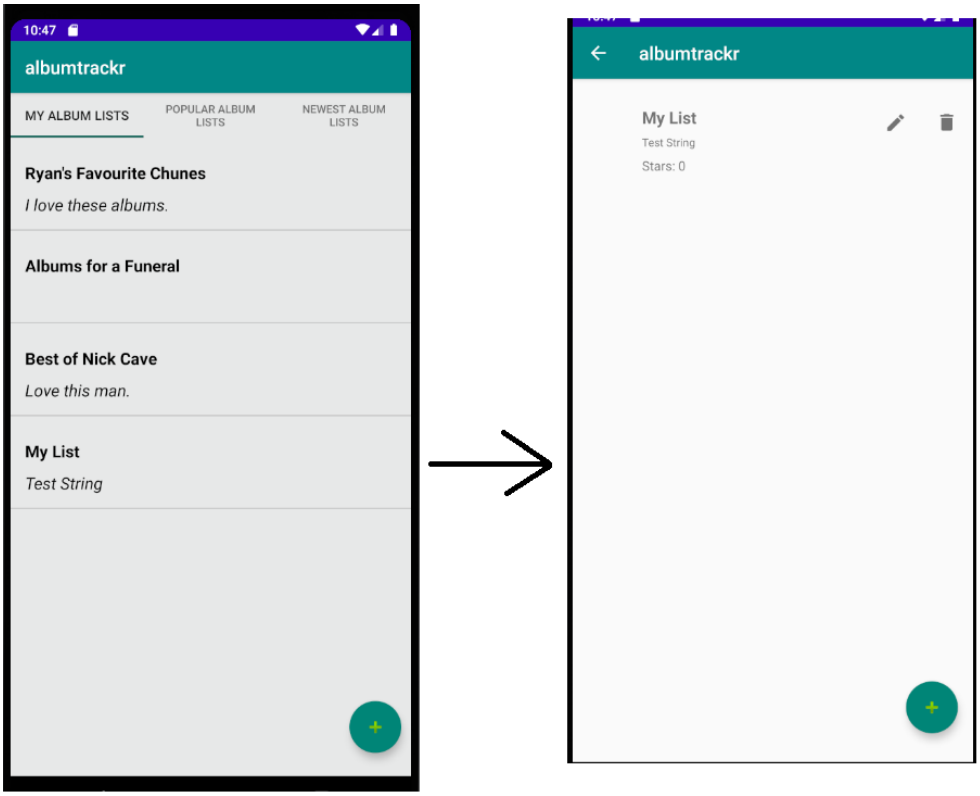


Application layout for viewing **AlbumLists**. The other fragments are similar, but just with different sorted data.

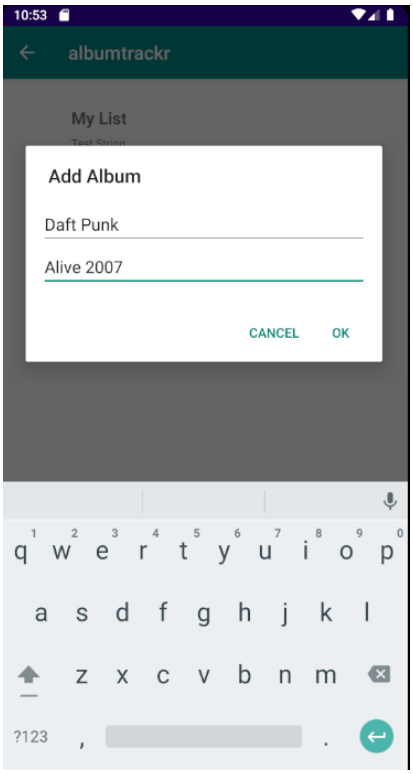
Adding an Album



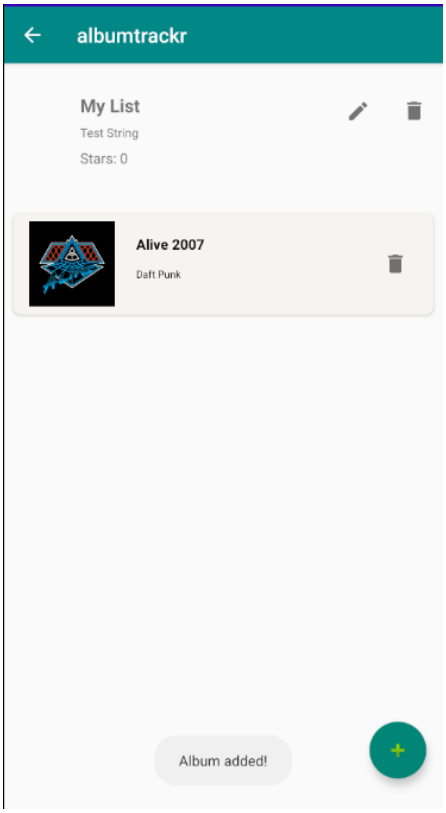
Showcasing the **Add Album List** dialog box. This pops up when the floating action button as seen previously is pressed. The album list name and description are passed to the API, which creates a list using a request containing JSON in the body of the request, adding the list. This will then refresh the activity, allowing the list to be visible when requests are made again.



Clicking the item on the list will bring you to our secondary activity – a view that displays the albums within the list. However, in our sample list none have been created. Let's add one.

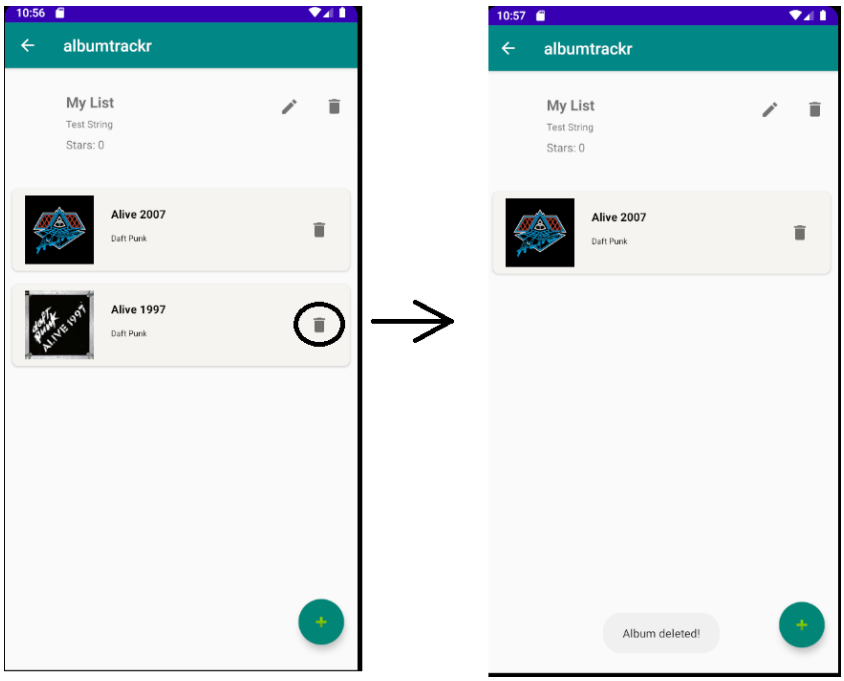


Clicking the floating action button brings up a dialog box which we can use to add an album to the list. It is worth noting again, **an album will only be added if it is in the Last.FM database**.



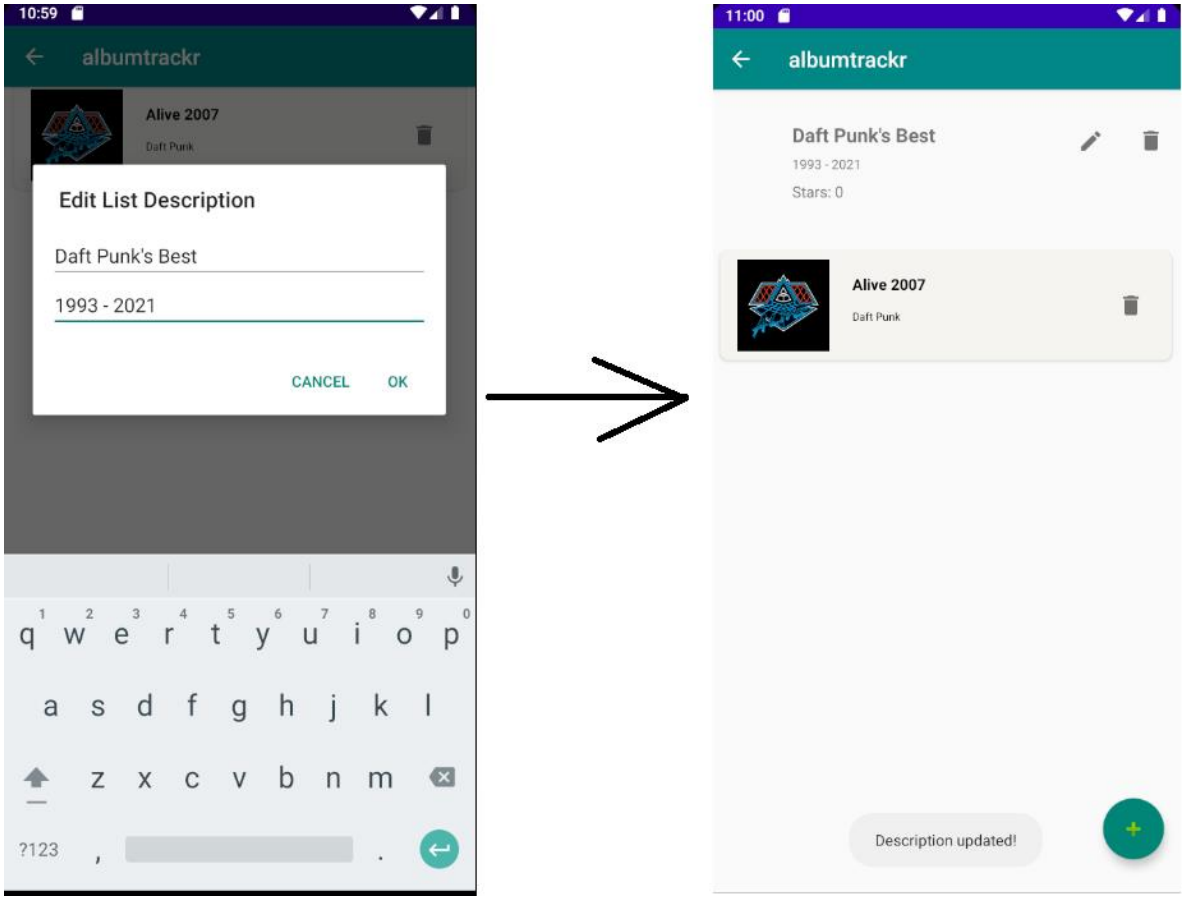
Our album is added, with the appropriate album artwork pulled from the Last.FM database.

Deleting and Album



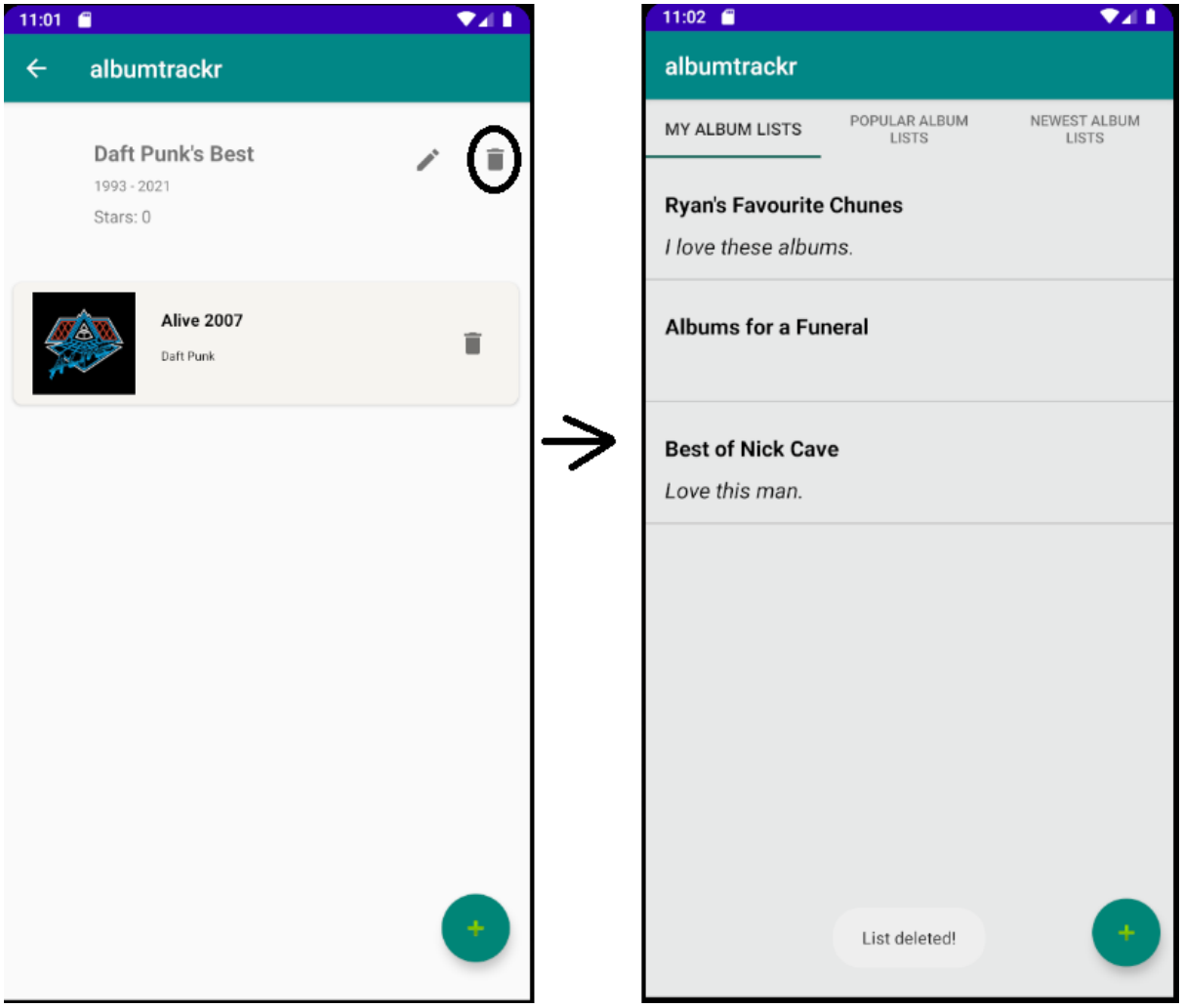
Let's say we added the wrong album and want to delete it from our list? We can. If you own the list, you can just press the trashcan icon to delete the album off your list. This is done making a DELETE query to the API.

Updating List Description



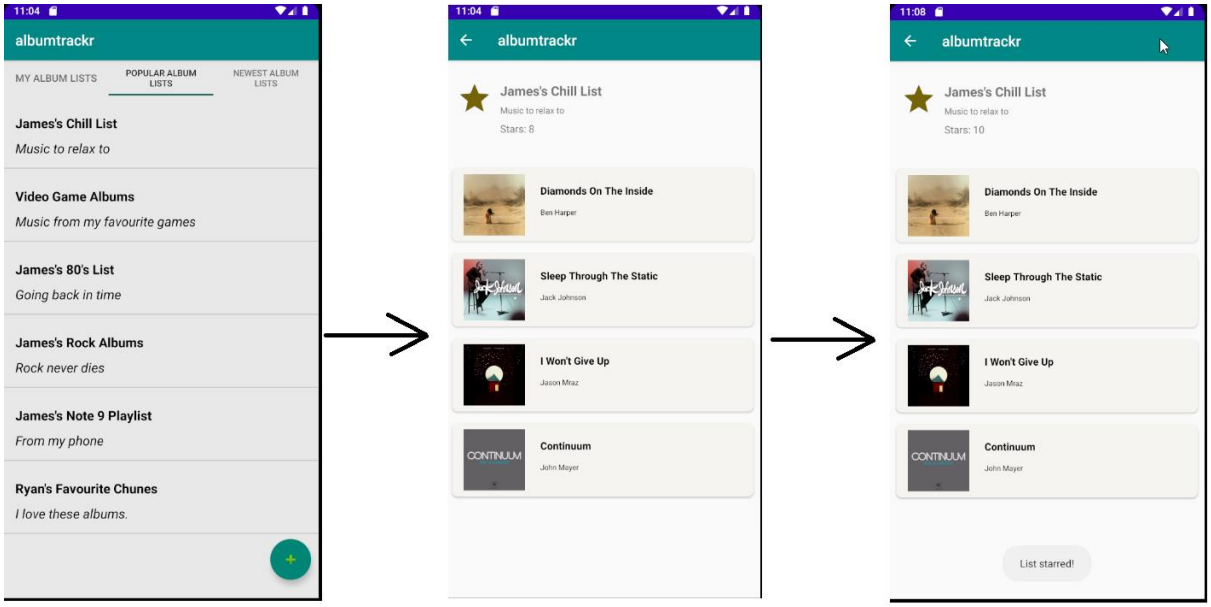
We can also update the name and description of our list. This is shown here, through our dialog box, making a PUT query to the API.

Deleting a List



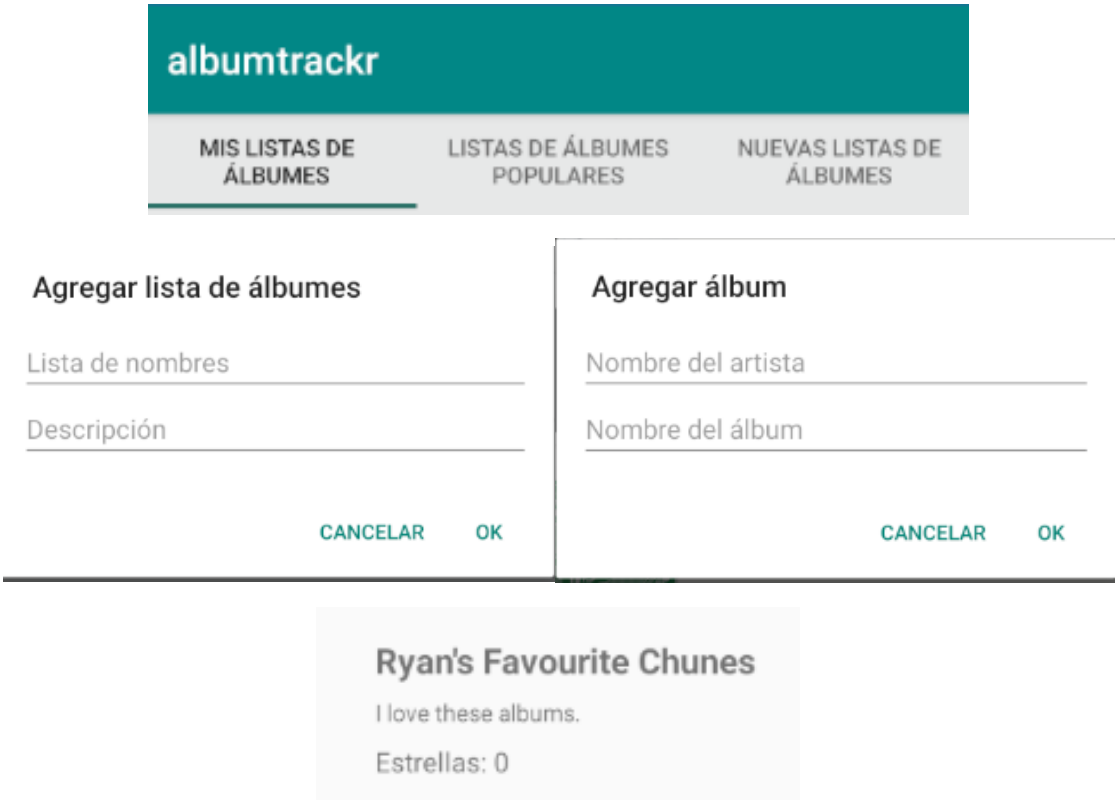
If we want to delete the list, we also can. This is done through the trashcan icon on the top of the list.

CRUD Permissions



As we can see, we cannot edit James' playlist, only he can do that from his phone because it shares the same username. We can only star lists that we do not own; we can not star our own list.

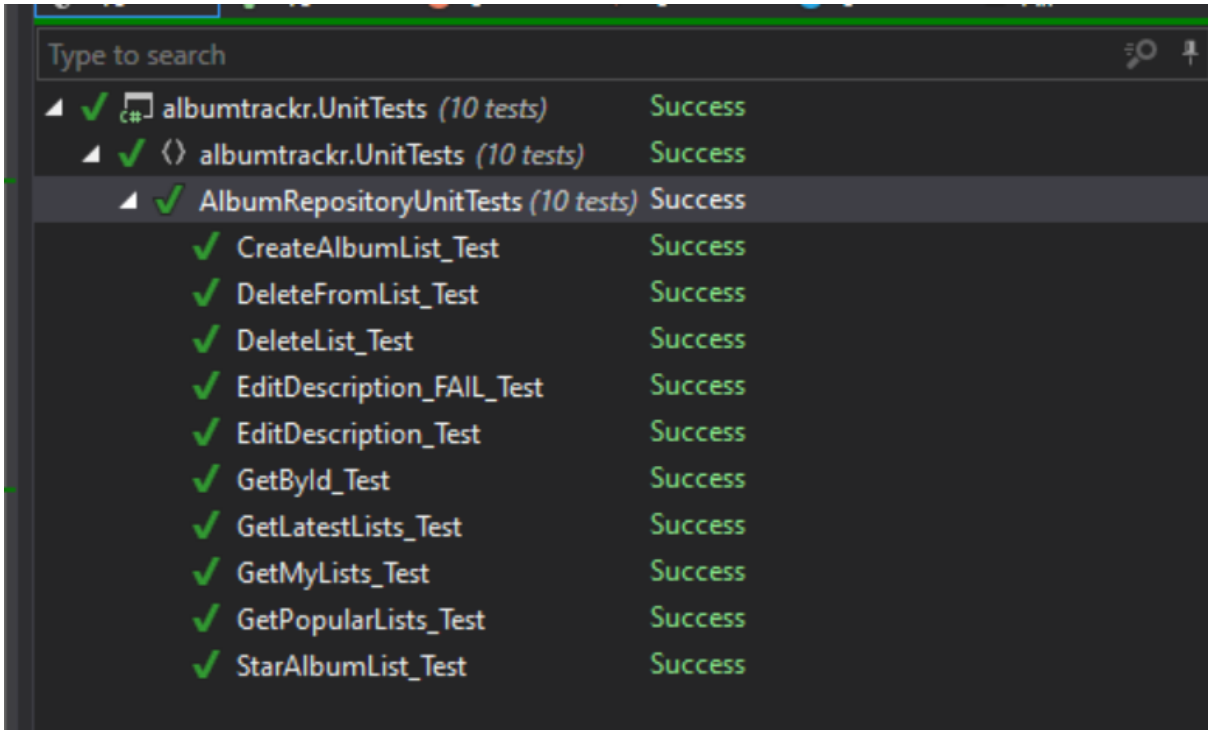
Internationalisation: We localized our app in Spanish, using the strings.xml file. Here is some examples of **albumtrackr** in Spanish.



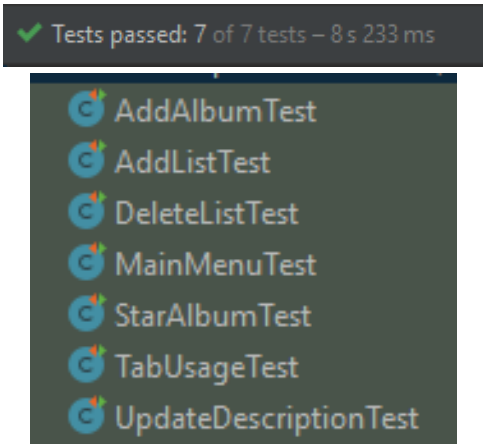
Testing

Our testing was conducted on two sides of the application. We configured a number of unit tests on both the front and back end of development. We also conducted espresso tests for Android. The tests are as follows:

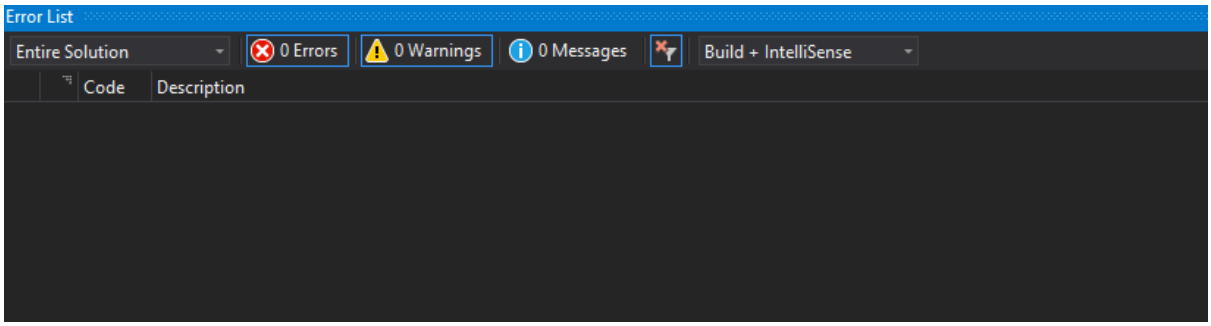
API Tests



Android Tests (Espresso)



Miscellaneous



SonarLint reports no issues with code quality. This is achieved thanks to the likes of ReSharper.

Conclusion

In conclusion, we believe we have learned a great deal about Android development and hope to carry these skills forward into the future. This CA was not only important as a learning tool, but as a valuable addition to our portfolios for future employers. We are sure that the things we have learned in the past few weeks will stand to us as we continue to grow as Software Developers.