Micro optimizaciones y buenas prácticas

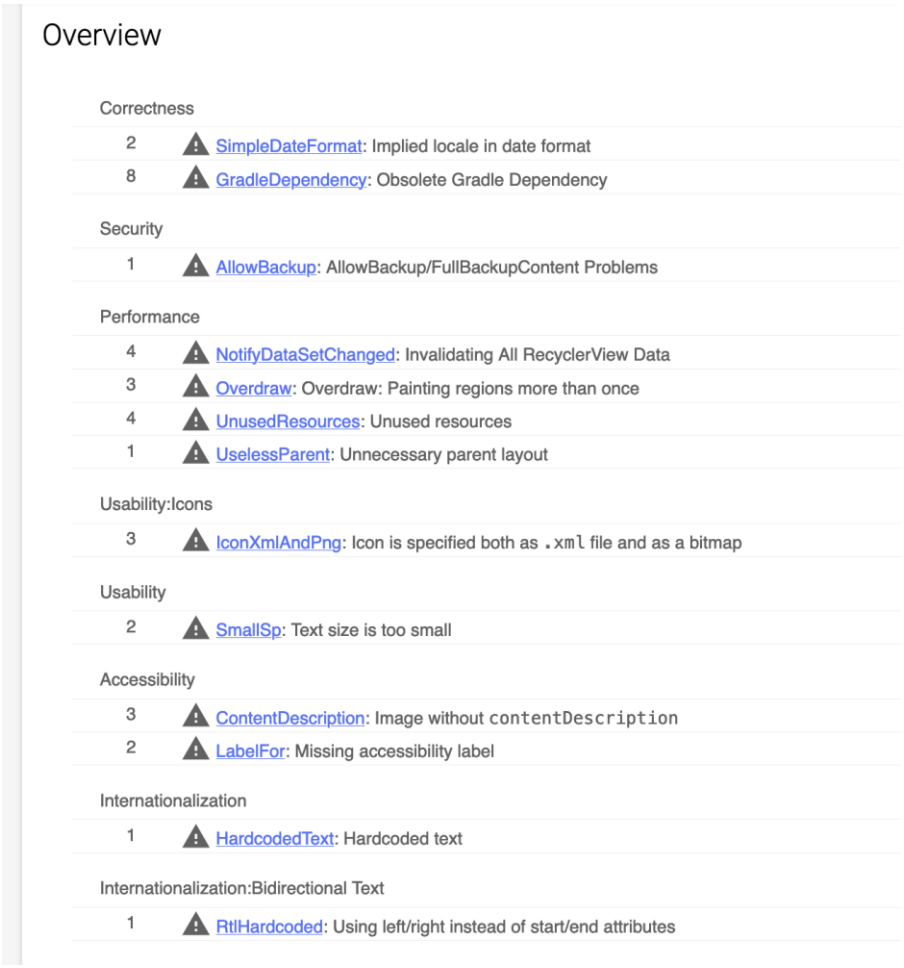
En el siguiente documento se presentan las diferentes buenas prácticas y micro optimizaciones que se realizaron en el proyecto para asegurar un buen desempeño del aplicativo.

Análisis estático

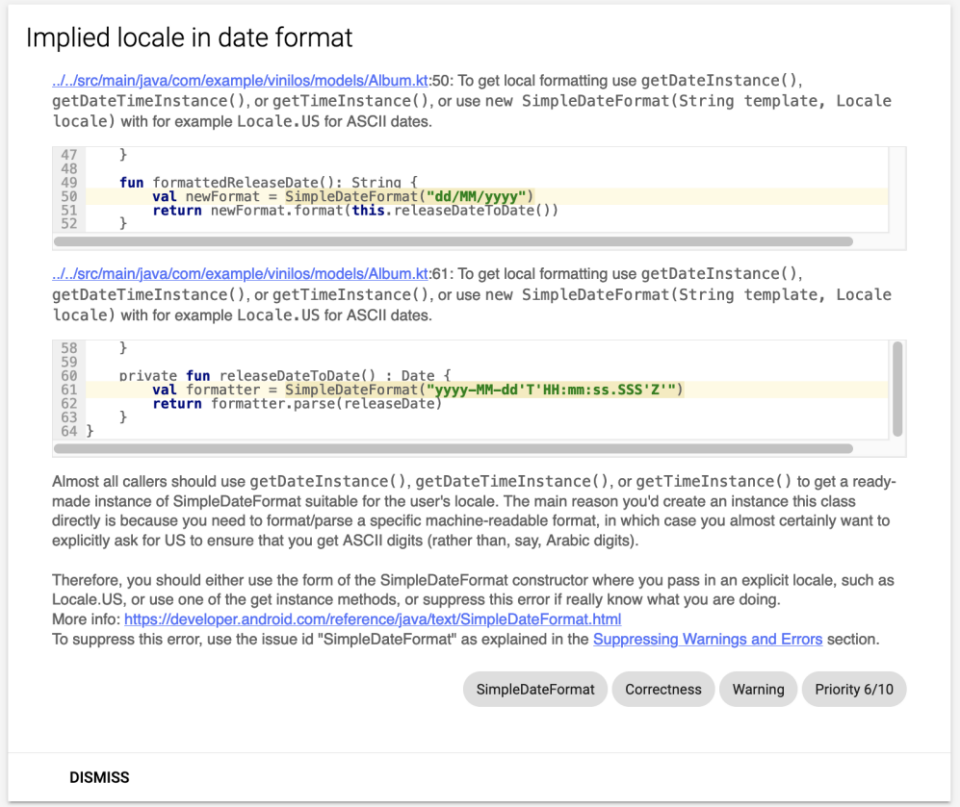
Para mejorar el código y asegurarnos que el código está bien estructurado, es confiable y eficiente, ejecutamos el Lint de Android por medio del comando:

./gradlew lint

Dicho comando genera un archivo .html con el reporte de hallazgos, issues y warnings a tomar en cuenta para mejorar el código. Este reporte presentó 35 warnings que involucran temas desde tamaños de imágenes hasta recursos sin utilizar:

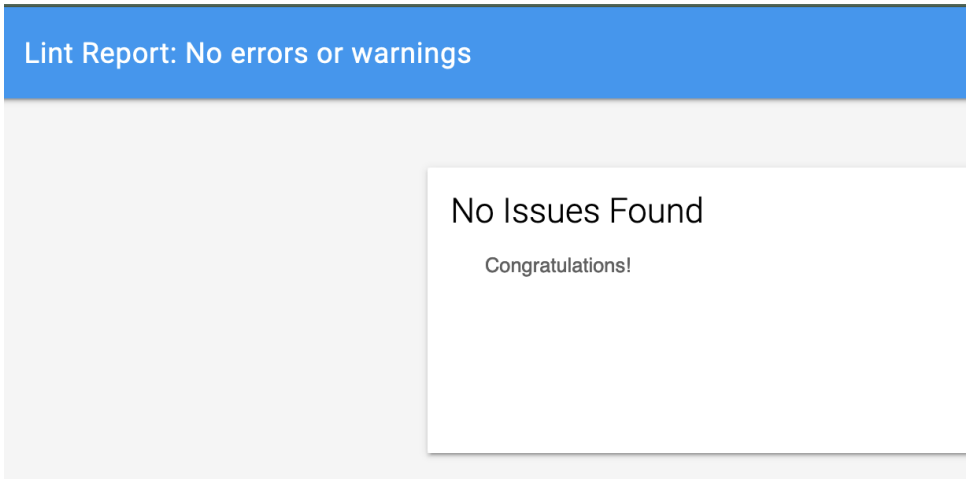


Además, este reporte presenta los lugares puntuales en el código donde ese encontraron issues y una explicación detallada de porque es un issue, por ejemplo, para las fechas:



El reporte inicial se puede encontrar en el repositorio: [Reporte inicial lint]

Tomando en cuenta el resultado del linter, se hicieron mejoras para reducir la cantidad de warnings, la siguiente imagen presenta el reporte luego de las mejoras:



Como se puede observar, la cantidad de warnings bajó a 0. En la mayoría de los casos la solución era intuitiva, por ejemplo, el tamaño del texto no podía ser menor a 11sp y el tamaño que usábamos era 10sp, así que lo aumentamos a 12sp para no alterar tanto la UI pero cumpliendo la regla. Adicionalmente, teníamos issues con recursos no utliizados en el archivo de strings y de colores por lo que se eliminaron, o, por otro lado, usábamos versiones pasadas en el gradle, entonces se actualizaron. Sin embargo, hubo 1 regla que no era muy claro cómo solucionarla, la referente a IconXmlAndPng, nos marcaba un warning porque teníamos íconos en formato .png y también en formato .xml, no obstante, tenemos diferentes definiciones de los íconos para que se adapten al tamaño de la pantalla, por lo que consideramos que el warning no aplica para este caso y se excluyó

en el gradle. Además, en lo que respecta a Overdraw, el lint marcaba 3 posibles layout que se estaban sobre dibujando, pero al analizar los layouts encontramos que no se estaban pintando varias veces y que era un falso positivo, entonces excluimos la regla de overdraw en esos layouts.

El reporte final se puede encontrar en el repositorio: Reporte final lint

Uso de corrutinas

Para evitar errores en el aplicativo de tipo ANR (Application not responding) se modificaron todos los métodos encargados de obtener u crear información en el servidor o base de datos local para que utilizaran corrutinas de Kotlin, estos servicios fueron los siguientes:

- GET – Obtener álbumes
- POST – Crear álbum
- GET - Obtener detalle de un álbum
- GET – Obtener artistas
- GET – Obtener coleccionistas

De los cuales representó cambios no solo en los ServiceAdapter sino también en los Repositorios y ViewModels. A continuación, se presenta un ejemplo de cómo fue la estructura cambiada para utilizar las corrutinas.

Anteriormente se contaba con un ServiceAdapter que hacía los llamados al servidor y para manejar de forma asincrónica la carga exitosa o fallida de datos utilizaban callbacks.

```
fun getAlbums(onComplete:(resp:List<Album>)->Unit, onError: (error: VolleyError)->Unit) {
    requestQueue.add(getRequest( path: "albums",  { response ->
            val resp = JSONArray(response)
            val list = Album.fromJSONArray(resp)
            onComplete(list)
        },
        { it: VolleyError!
            onError(it)
        }))
}
```

Por lo cual se procede a declarar la función como una suspend function y a utilizar el context de la corrutina en vez de los callback que fueron removidos.

```
suspend fun getAlbums() = suspendCoroutine<List<Album>>{ cont ->
    requestQueue.add(
        getRequest( path: "Albums",  { response ->
            val resp = JSONArray(response)
            val list = Album.fromJSONArray(resp)
            cont.resume(list)
        }, { it: VolleyError!
            cont.resumeWithException(it)
        })
    )
}
```

Seguido a esto se procede a cambiar el repositorio ya que anteriormente llama el ServiceAdapter pasándole los callback que recibe por parámetro.

```
fun refreshData(callback: (List<Album>)->Unit, onError: (VolleyError)->Unit) {
    AlbumServiceAdapter.getInstance(application).getAlbums({ it: List<Album>
        callback(it)
    }, onError)
}
```

Este cambio requiere declarar que refresh data ahora es una suspend function ya que una suspend function solo puede ser utilizada desde otra de tipo suspend o desde un scope lifecycle o ViewModel.

```
suspend fun refreshData(): List<Album> {
    return AlbumServiceAdapter.getInstance(application).getAlbums();
}
```

Finalmente, el ViewModel hace uso del refresh data por lo cual es necesario cambiar su implementación para que utilice las funciones suspend. Anteriormente pasaba los callback por parámetro como se ve a continuación.

```
fun refreshDataFromNetwork() {
    _isLoading.value = true
    _isNetworkErrorShown.value = false
    albumsRepository.refreshData({ it: List<Album>
        _albums.postValue(it)
        initialAlbums = it
        _isLoading.value = false
    },{ it: VolleyError
        _isNetworkErrorShown.value = true
        _isLoading.value = false
    })
}
```

Una vez realizado el cambio se utiliza el *viewModelScope* y un *try/catch*. Es importante recordar que ya no es necesario utilizar callbacks y para hacer cualquier cambio en el *withContext* es necesario utiliza el *postValue*.

```
fun refreshDataFromNetwork() {
    _isLoading.value = true
    _isNetworkErrorShown.value = false
    viewModelScope.launch(Dispatchers.Default) { this: CoroutineScope
        withContext(Dispatchers.IO){ this: CoroutineScope
            try {
                val data = albumsRepository.refreshData()
                _albums.postValue(data)
                initialAlbums = data
                _isLoading.postValue( value: false)
            } catch(e: Exception) {
                _isNetworkErrorShown.postValue( value: true)
                _isLoading.postValue( value: false)
            }
        }
    }
}
```

Los archivos modificados fueron los siguientes:

- AlbumsViewModel.kt
- AlbumViewModel.kt
- ArtistViewModel.kt
- CollectorViewModel.kt
- CreateAlbumViewModel.kt
- AlbumRepository.kt
- ArtistRepository.kt
- CollectorRepository.kt
- AlbumServiceAdapter.kt
- ArtistServiceAdapter.kt
- CollectorServiceAdapter.kt

Micro optimizaciones de uso de memoria

Previamente se hace uso de un HashMap para almacenar los extras que van a ser enviados por medio de un activity.

```
btn_sign_as_user.setOnClickListener { it: View!
    val extras: HashMap<String, String> = hashMapOf("userType" to "user")
    goToActivity(MenuActivity::class.java, extras = extras)
}
btn_sign_as_collector.setOnClickListener { it: View!
    val extras: HashMap<String, String> = hashMapOf("userType" to "collector")
    goToActivity(MenuActivity::class.java, extras = extras)
}
```

Se hace la migración para que ahora utilicen en vez un ArrayMap y así optimizar el uso de memoria.

```
btn_sign_as_user.setOnClickListener { it: View!
    val extras: ArrayMap<String, String> = ArrayMap()
    extras["userType"] = "user"
    goToActivity(MenuActivity::class.java, extras = extras)
}
btn_sign_as_collector.setOnClickListener { it: View!
    val extras: ArrayMap<String, String> = ArrayMap()
    extras["userType"] = "collector"
    goToActivity(MenuActivity::class.java, extras = extras)
}
```

Por otro lado, se instanciaban objetos dentro de ciclos, por lo que luego de cada iteración el objeto quedaba inalcanzable.

```
for (i in 0 until resp.length()) {
    val item = resp.getJSONObject(i)
    list.add(i, Artist(artistId = item.getInt( name: "id"),name = item.getString( name: "name"), image = item.getString( name: "image")
}
```

Se hace la migración para que ahora se cree la variable fuera del ciclo, se reutilice la variable, evitar que se creen múltiples objetos y así optimizar el uso de memoria.

```
var item: JSONObject
for (i in 0 until resp.length()) {
    item = resp.getJSONObject(i)
    list.add(i, Artist(artistId = item.getInt( name: "id"),name = item.getString( name: "name")
}
```

Uso de almacenamiento local

Para disminuir la carga en la app y fomentar la conectividad eventual se procedió a utilizar una base de datos local gracias a la librería Room por lo cual se modificaron los modelos para que fueran entidades.

```
@Entity(tableName = "albums_table")
data class Album (
    @PrimaryKey val albumId: Int,
    val name: String,
    val cover: String,
    val releaseDate: String,
    val description: String,
    val genre: String,
    val recordLabel: String,
    val performers: List<Performer> = mutableListOf<Performer>(),
    val tracks: List<Track> = mutableListOf<Track>(),
    var createdAt: Long = System.currentTimeMillis()
) {
```

Luego se procedió a agregar a crear los correspondientes DAOs que se encargaran de los queries a la base de datos.

```kotlin
@Dao
interface AlbumDAO {
    @Query("SELECT * FROM albums_table ORDER BY createdAt ASC")
    fun getAlbums(): List<Album>

    @Query("SELECT * FROM albums_table WHERE albumId = :id")
    fun getAlbum(id: Int): List<Album>

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(album: Album)

    @Query("DELETE FROM albums_table")
    suspend fun deleteAll(): Int
}
```

Finalmente se modificó los repository para que utilicen cache en caso de tenerlo, por lo cual luego de un primer request se utilizara el cache disminuyendo la carga de red.

```kotlin
suspend fun refreshData(): List<Album> {
    val db = VinylRoomDatabase.getDatabase(application.applicationContext)
    val cached = db.albumDao().getAlbums()
    return if(cached.isNullOrEmpty()) {
        val albums = AlbumServiceAdapter.getInstance(application).getAlbums()
        for(album in albums) {
            db.albumDao().insert(album)
        }
        albums
    } else {
        cached
    }
}
```

Cache de imágenes

Para mejorar el performance de la carga de imágenes en el aplicativo se procede a modificar el BindingAdapter existente para que utilice cache.

```kotlin
@BindingAdapter("app:imageUri")
fun loadImageWithUri(imageView: ShapeableImageView, imageUri: String) {
    Glide.with(imageView.context).load(
        Uri.parse(imageUri)
    ).into(imageView)
}
```

Se procede a utilizar el método diskCacheStrategy, se intentó utilizar *centerCrop* y *asBitmap* para disminuir el tamaño de las imágenes sin embargo se encontró que las cargas de imágenes incrementaban, por lo cual solo se incluye el cache.

```
@BindingAdapter("app:imageUri")
fun loadImageWithUri(imageView: ShapeableImageView, imageUri: String) {
    Glide.with(imageView.context).load(
        Uri.parse(imageUri)
    ).diskCacheStrategy(DiskCacheStrategy.AUTOMATIC)
        .into(imageView)
}
```

Perfilamiento

Para analizar el uso de recursos y rendimiento de la aplicación, se usaron herramientas de Android Studio para generar un reporte de perfilamiento, el cual incluye información de uso de CPU, Memoria, Red y Energía. Dicho reporte se generó en 3 dispositivos físicos listados a continuación:

Id dispositivo	Marca	Versión Android	RAM
1	Oneplus 6T	11	8 GB
2	Redmi Note 9	10	4 GB
3	Galaxy A30s	11	4 GB

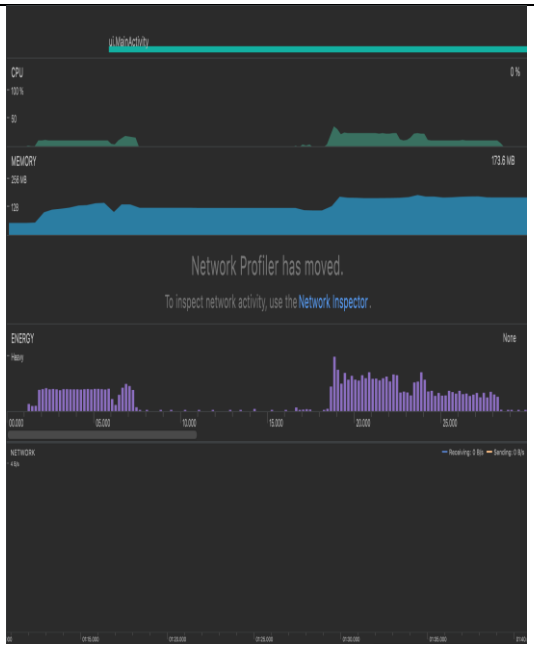
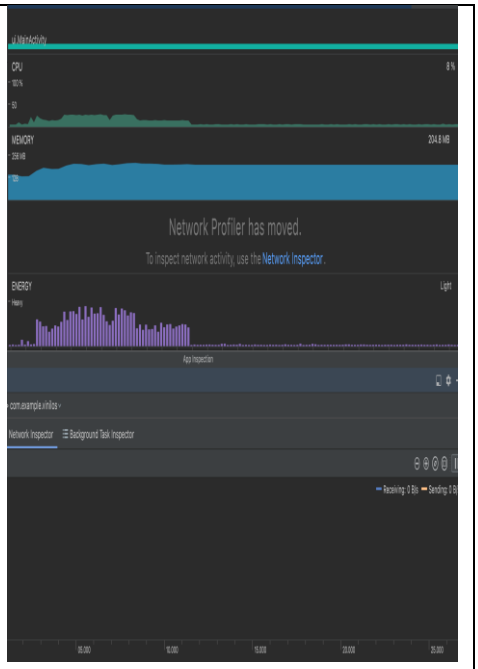
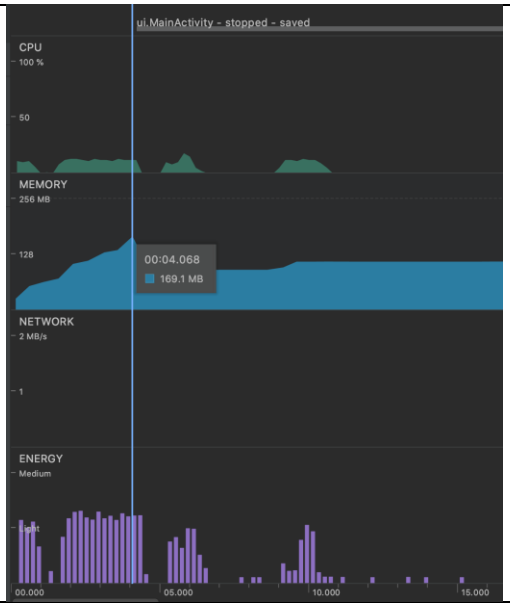
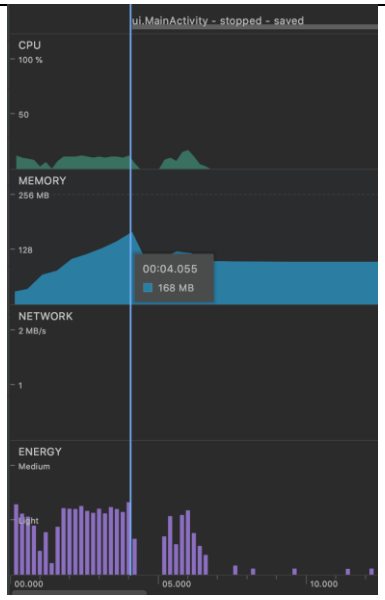
Además, se generó el reporte en dos momentos, la primera vez antes de realizar micro optimizaciones y buenas prácticas, y la segunda vez luego de que se implementaron micro optimizaciones, buenas prácticas, uso de caché y corrutinas, de tal forma que es posible comparar un antes y un después en el rendimiento de la app.

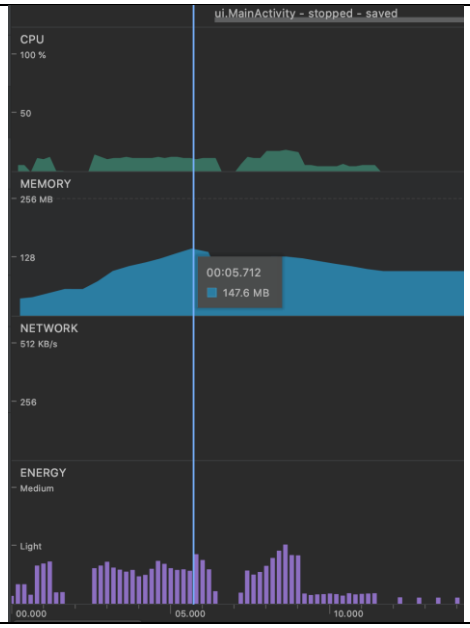
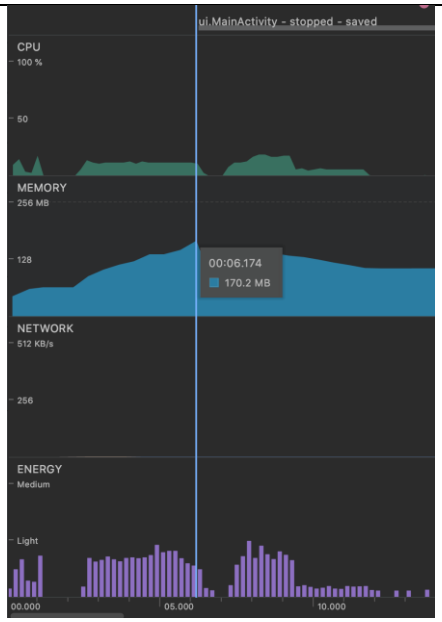
A continuación, se muestra el perfilamiento antes y después de las micro optimizaciones y por cada una de las historias de usuario implementadas:

Home

En la tabla se puede observar el antes y después cuando se abre la aplicación por primera vez, como se puede ver, en todos los dispositivos el comportamiento es similar, los picos son similares y el consumo de recursos también. Esto es esperado dado que no se aplicó ninguna optimización para iniciar la aplicación.

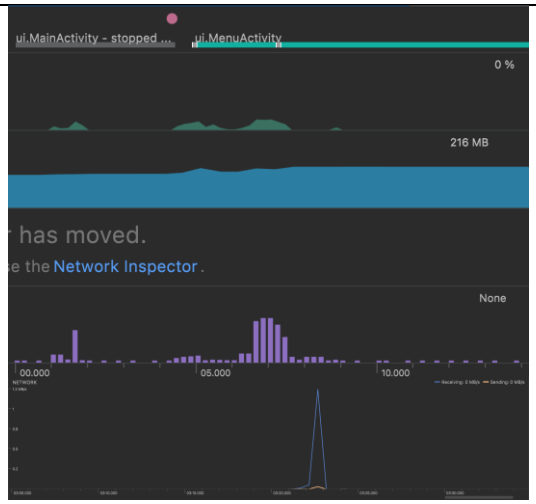
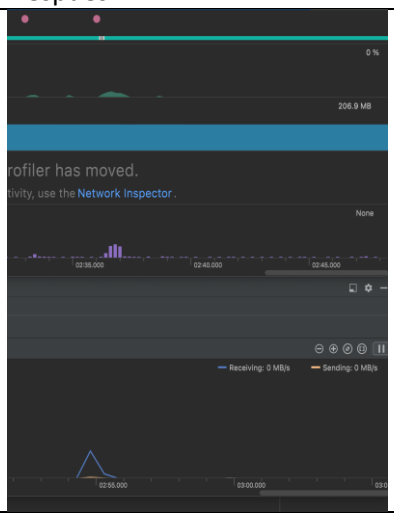
Dispositivo	Antes	Después

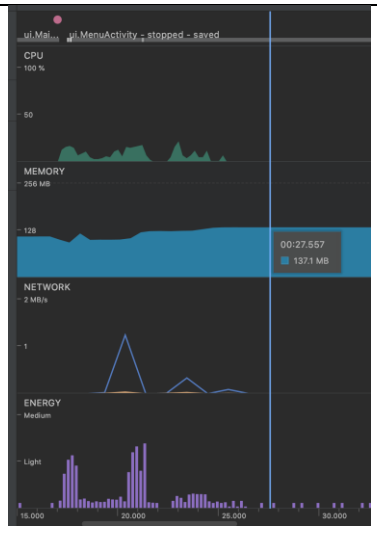
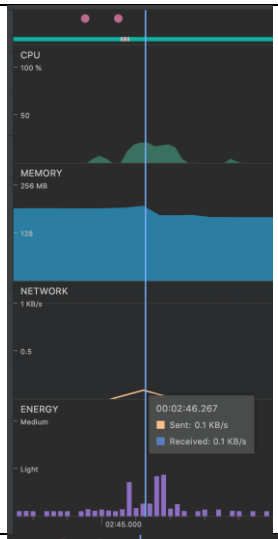
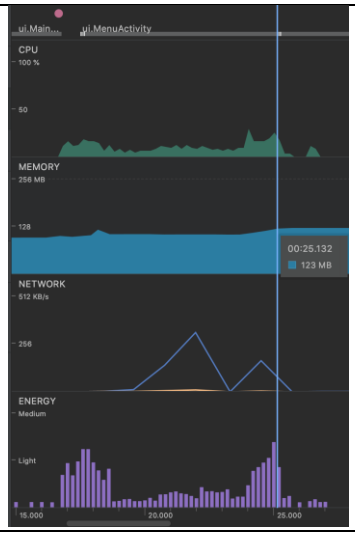
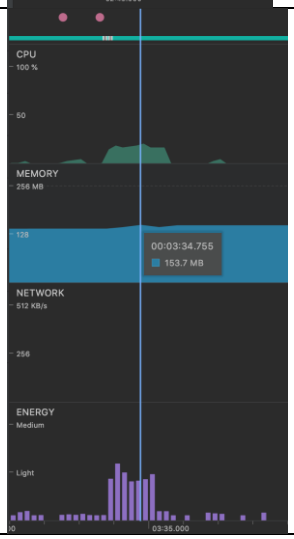
1	 ui.MainActivity CPU MEMORY Network Profiler has moved. To inspect network activity, use the Network Inspector . ENERGY NETWORK	 ui.MainActivity CPU MEMORY Network Profiler has moved. To inspect network activity, use the Network Inspector . ENERGY Network Inspector Background Task Inspector
2	 ui.MainActivity - stopped - saved CPU 100 % 50 MEMORY 256 MB 128 00:04.068 169.1 MB NETWORK 2 MB/s 1 ENERGY Medium 00.000 05.000 10.000 15.000	 ui.MainActivity - stopped - saved CPU 100 % 50 MEMORY 256 MB 128 00:04.055 168 MB NETWORK 2 MB/s 1 ENERGY Medium 00.000 05.000 10.000

3		

HU01 – Consultar catálogo de álbumes

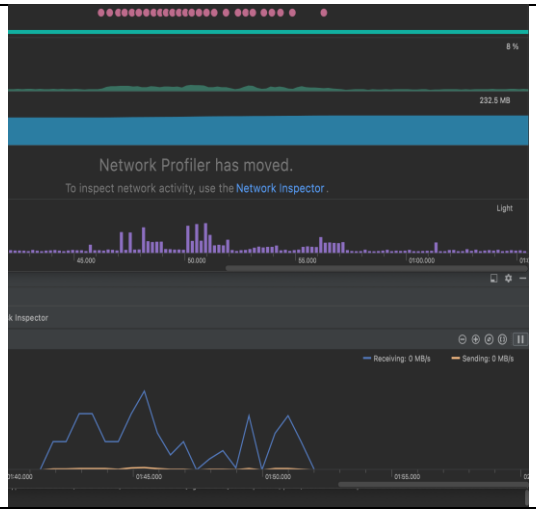
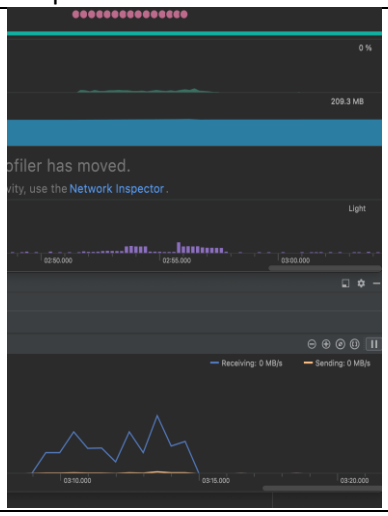
En el caso de consultar el catálogo de álbumes, se puede ver que hubo un cambio antes y después de las optimizaciones dado que ahora se usa caché para mantener la información sin tener que hacer consultas al backend, es por esto que la cantidad de conexiones es más baja después comparado con el antes, pero el uso de memoria incrementa.

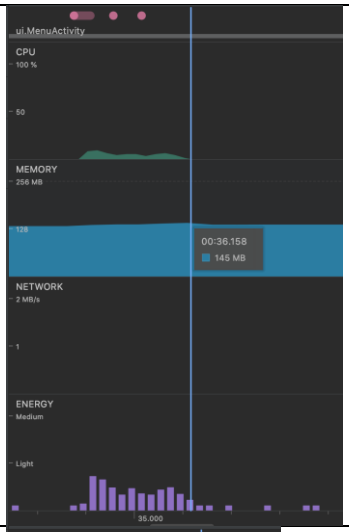
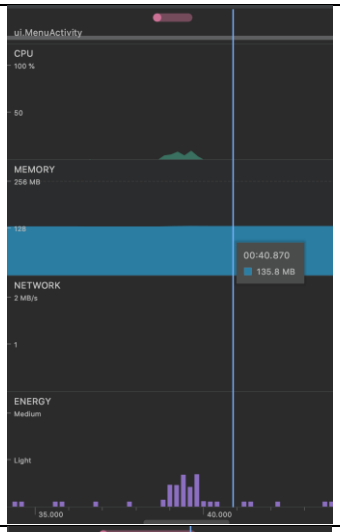
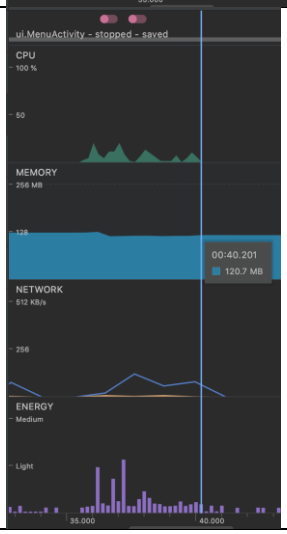
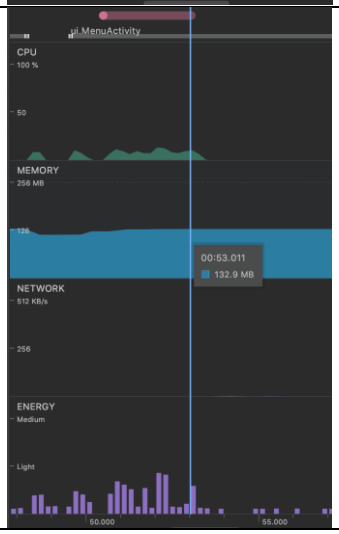
Dispositivo	Antes	Después
1		

2		
3		

HU01 – Consultar catálogo de álbumes, haciendo scroll en el listado

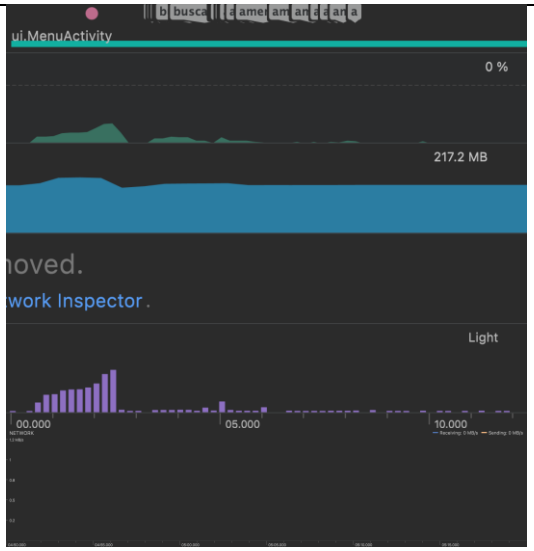
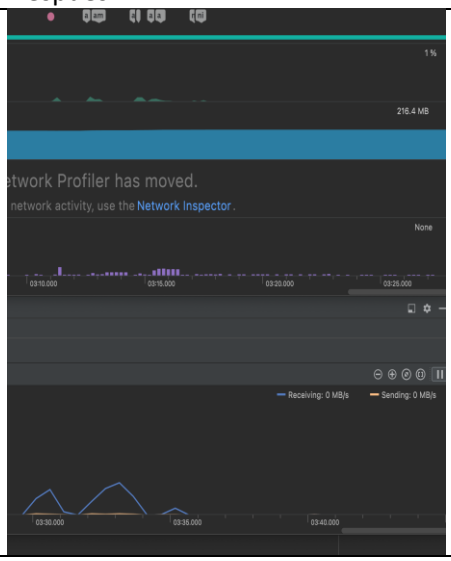
En el caso de hacer scroll en el listado de álbumes, no se ve mayor diferencia entre el antes y el después.

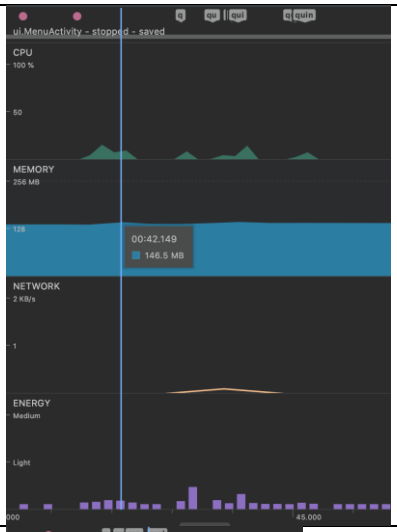
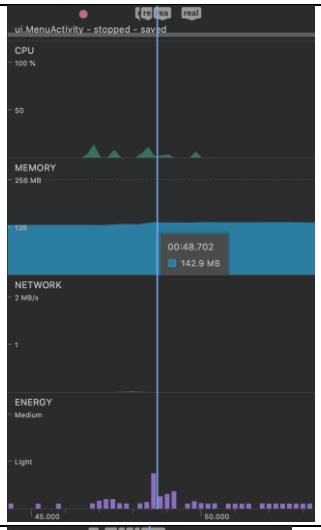
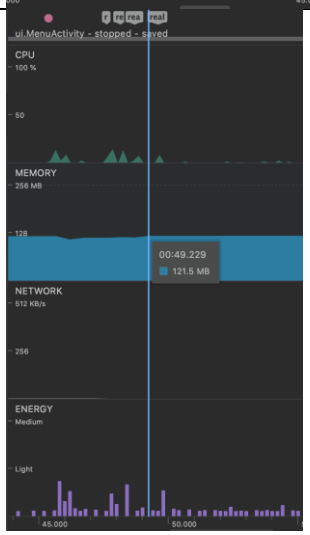
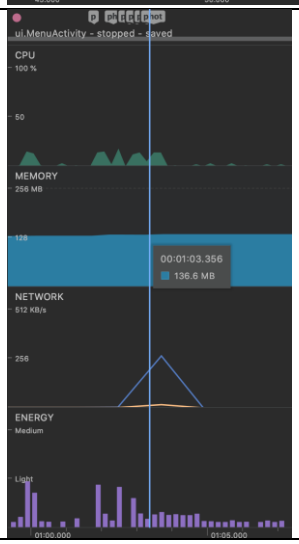
Dispositivo	Antes	Después
1		

2		
3		

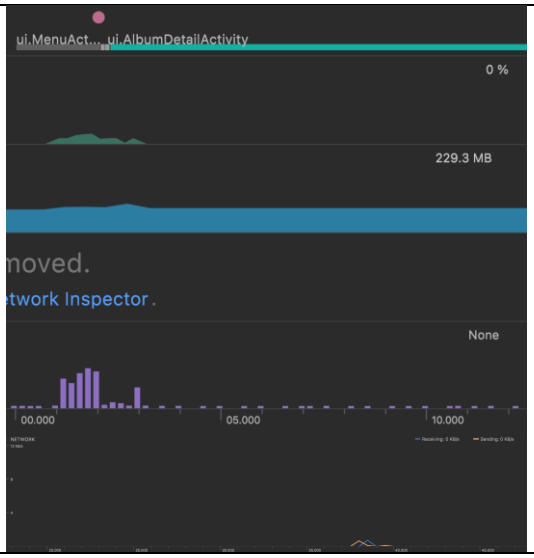
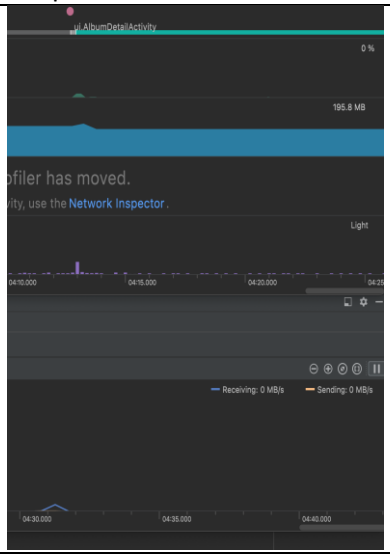
HU01 – Consultar catálogo de álbumes, filtrando por nombre

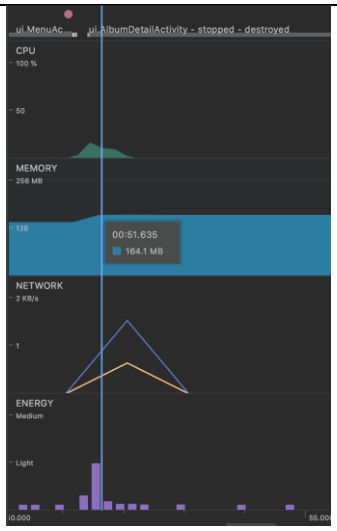
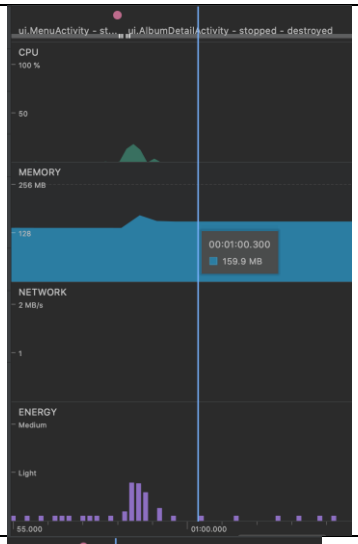
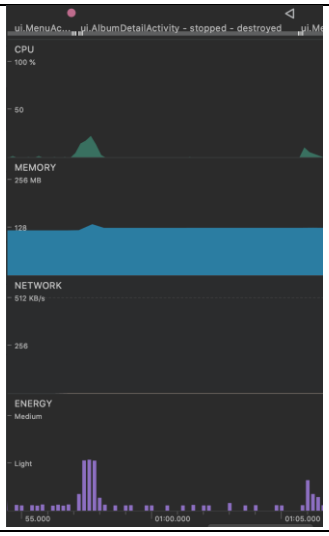
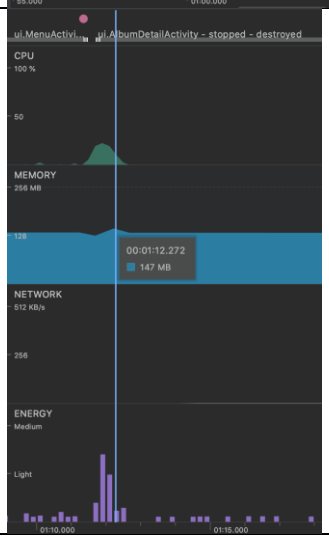
Similar al punto anterior, filtrando por nombre tampoco hay mayor diferencia entre antes y después de las optimizaciones

Dispositivo	Antes	Después
1		

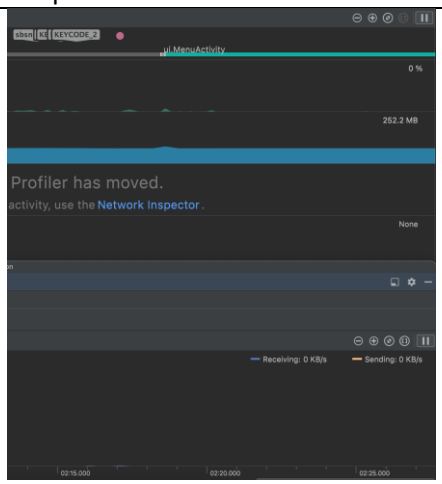
2		
3		

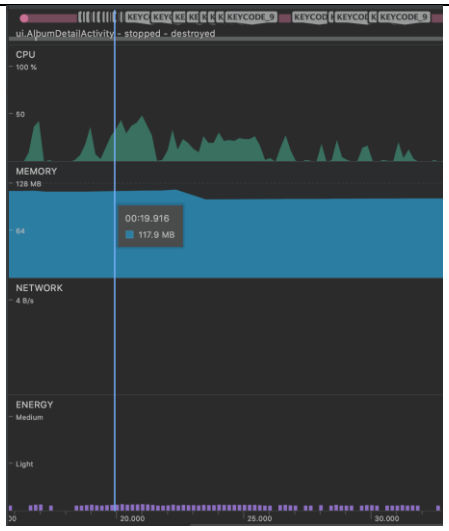
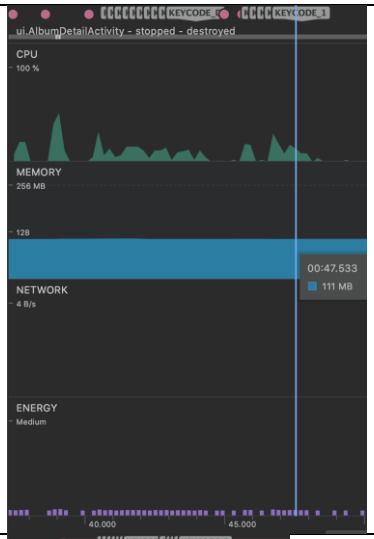
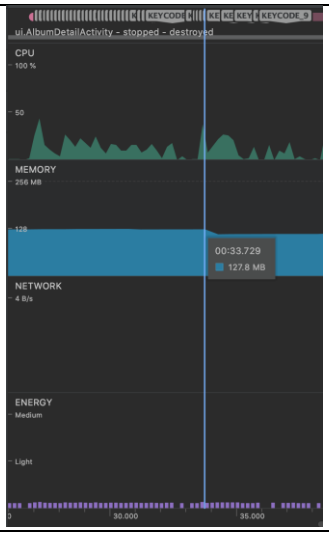
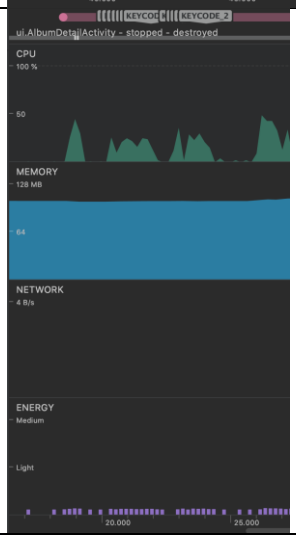
HU02 – Consultar detalle de un álbum

Dispositivo	Antes	Después
1		

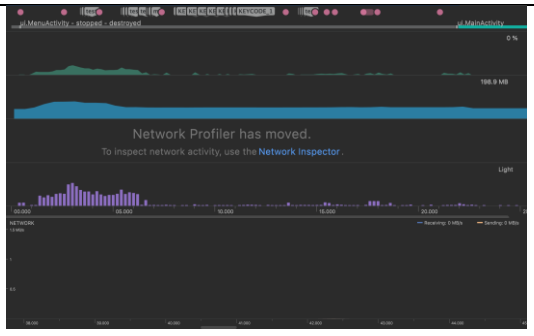
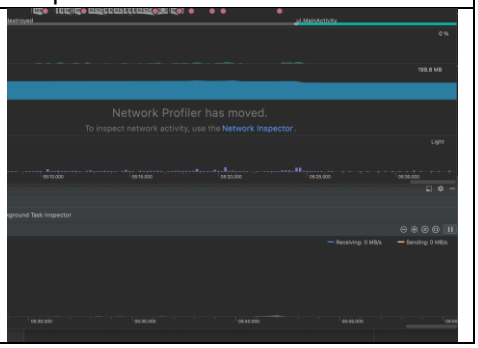
2		
3		

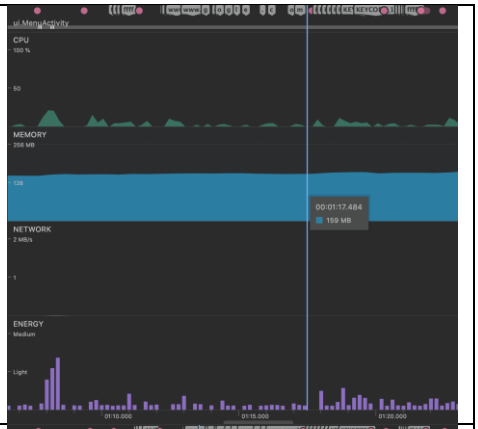
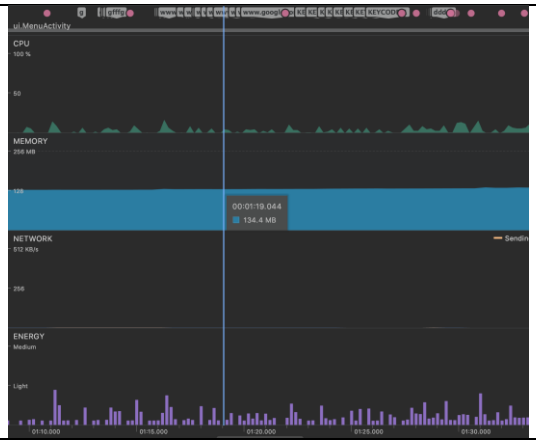
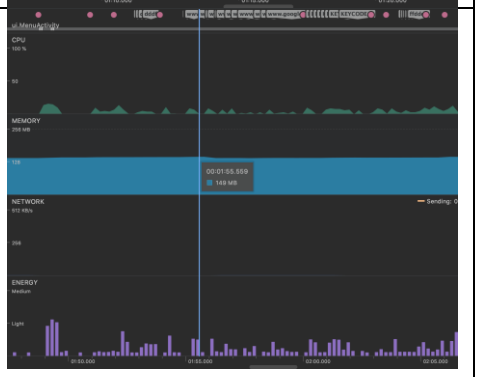
HU08 – Asociar track a un álbum

Dispositivo	Antes	Después
1		

2		
3		

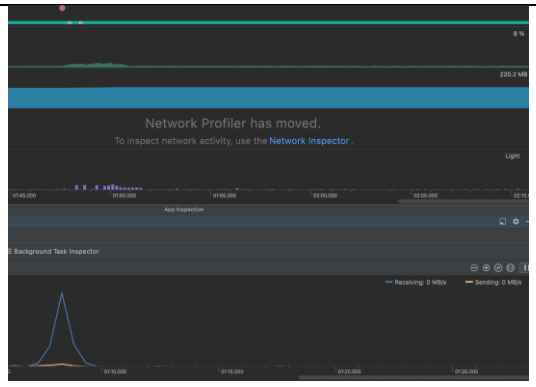
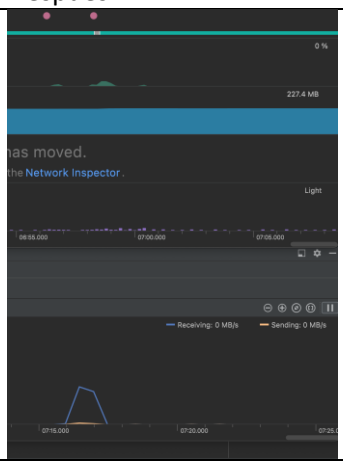
HU07 – Crear un álbum

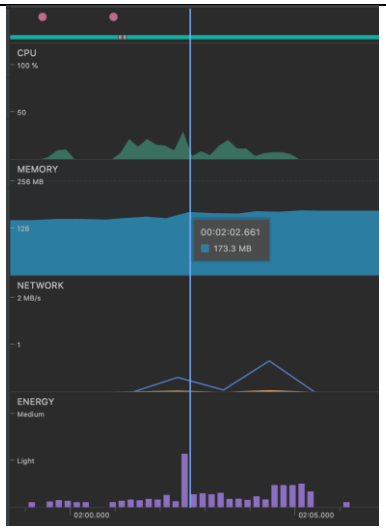
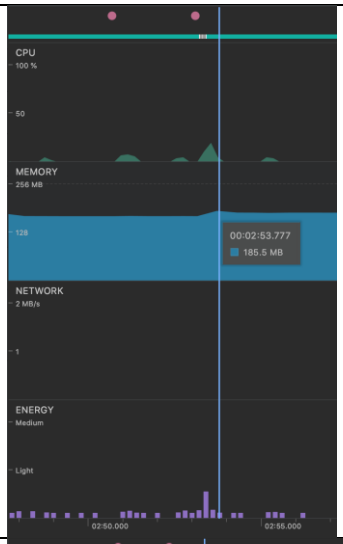
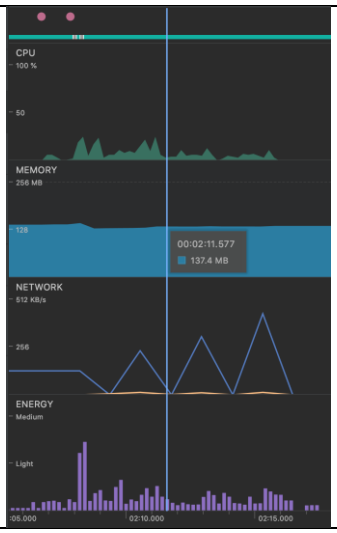
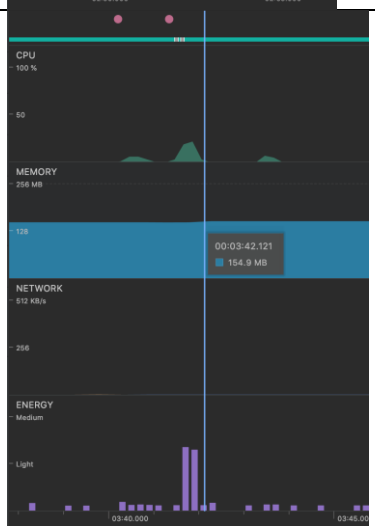
Dispositivo	Antes	Después
1		

2		
3		

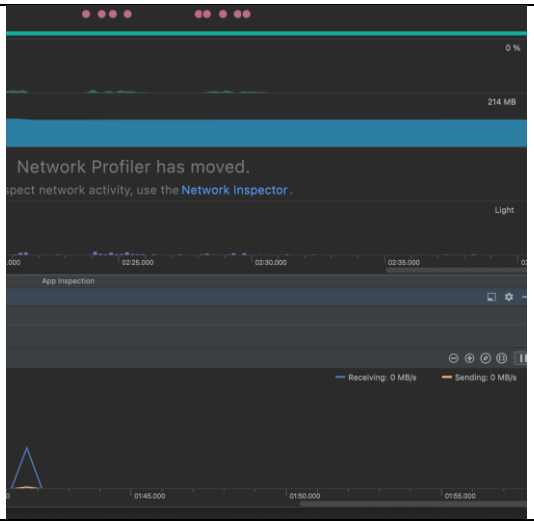
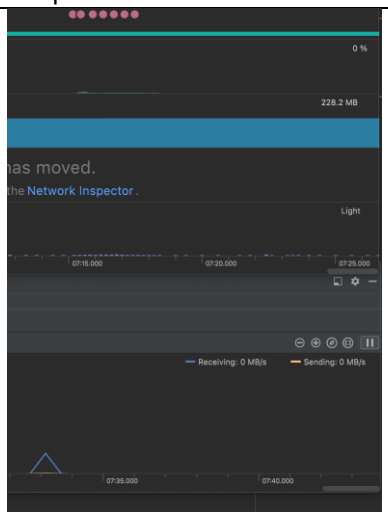
HU03 – Consultar listado de artistas

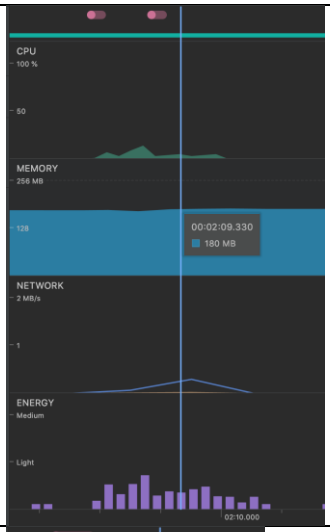
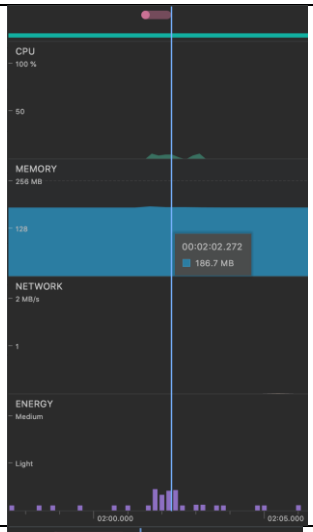
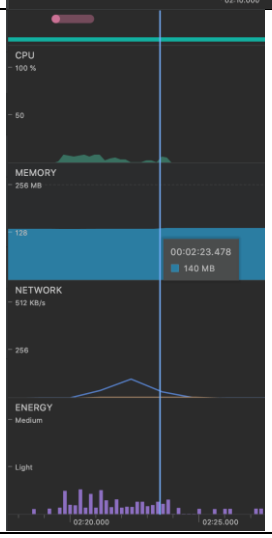
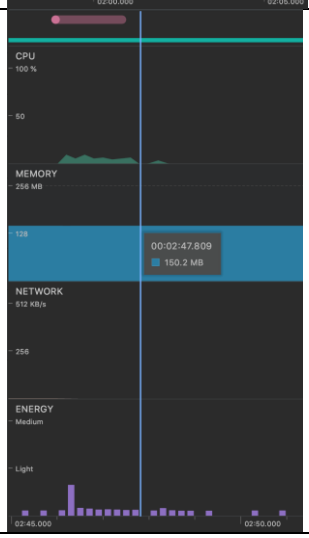
En el caso de consultar el catálogo de artistas, se puede ver que hubo un cambio antes y después de las optimizaciones dado que ahora se usa caché para mantener la información sin tener que hacer consultas al backend, es por esto que la cantidad de conexiones es más baja después comparado con el antes, pero el uso de memoria incrementa.

Dispositivo	Antes	Después
1		

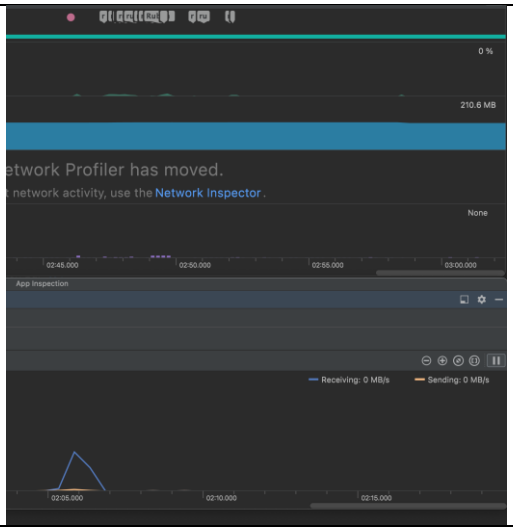
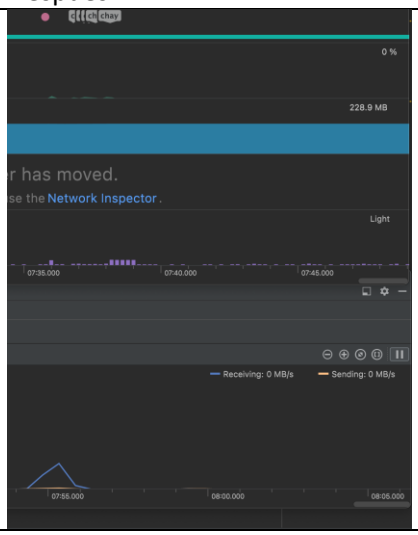
2		
3		

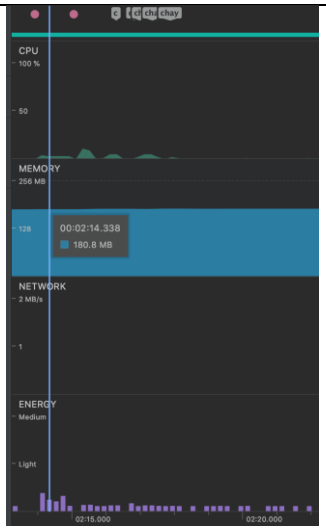
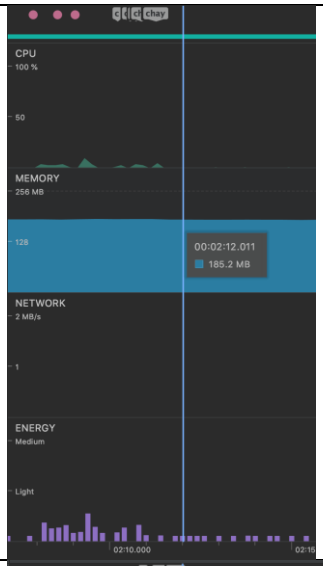
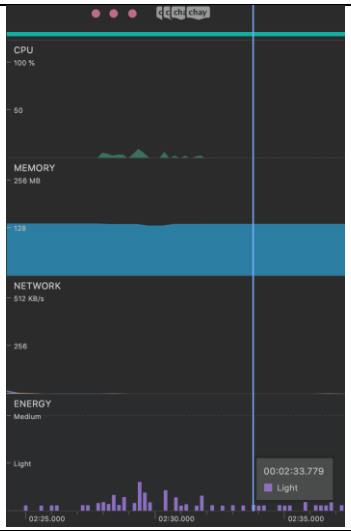
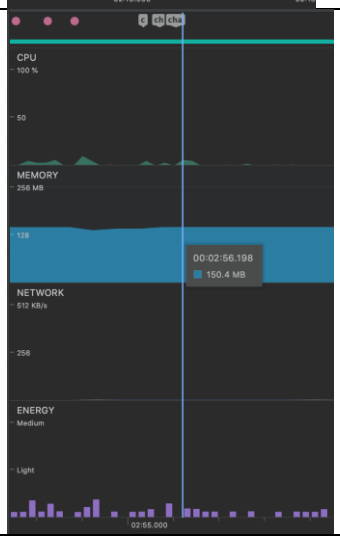
HU03 – Consultar listado de artistas, haciendo scroll en el listado

Dispositivo	Antes	Después
1		

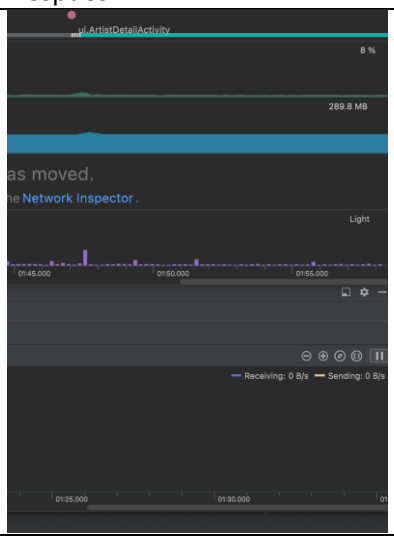
2		
3		

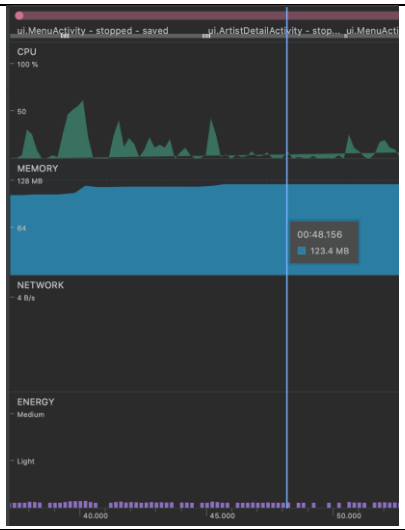
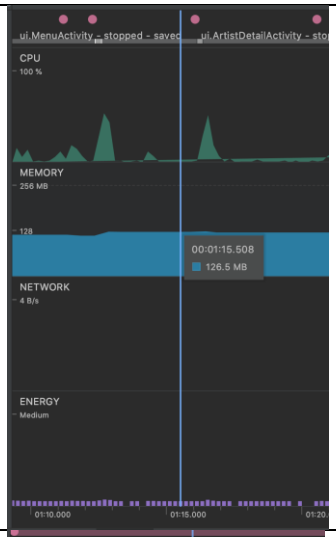
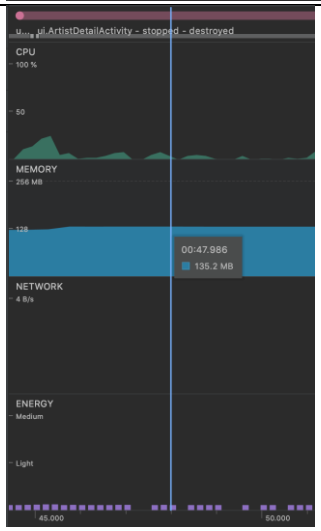
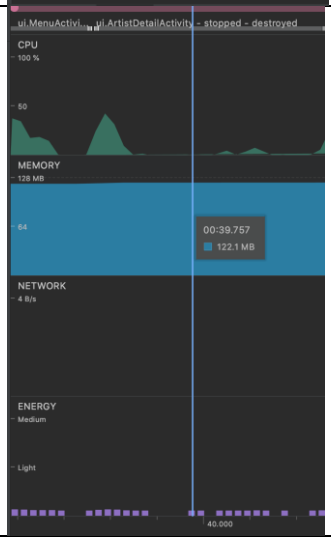
HU03 – Consultar listado de artistas, filtrando por nombre

Dispositivo	Antes	Después
1		

2		
3		

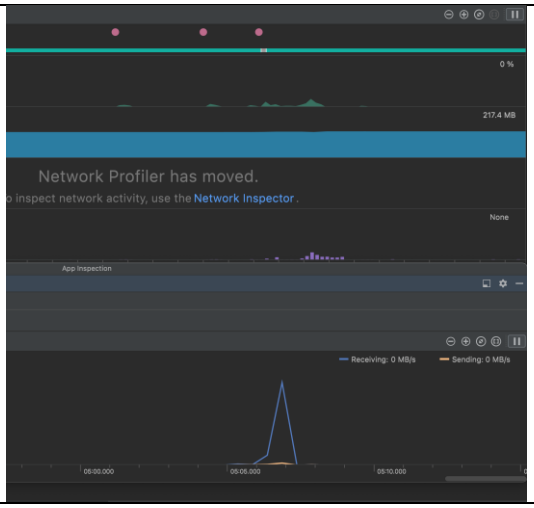
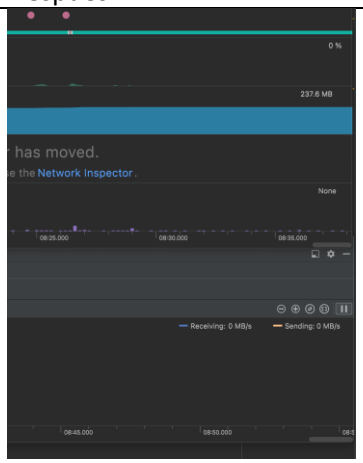
HU04 – Consultar detalle de artista

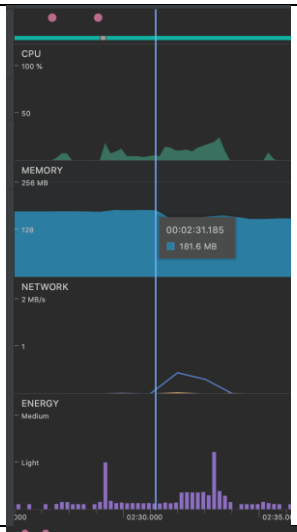
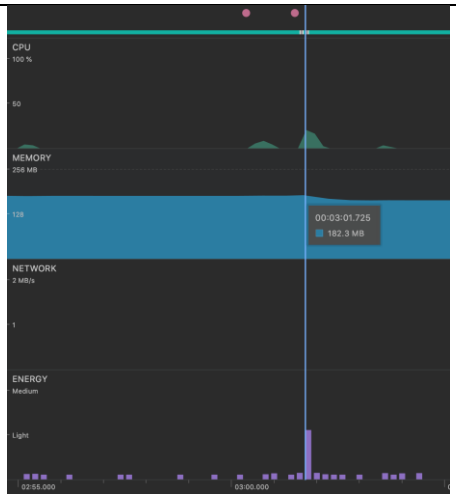
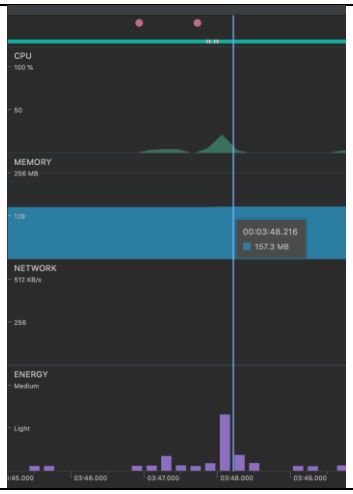
Dispositivo	Antes	Después
1		

2		
3		

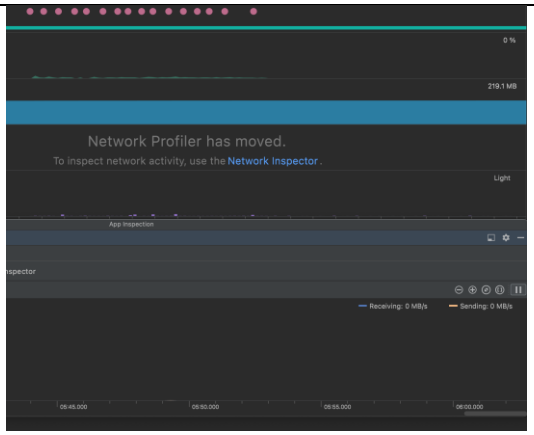
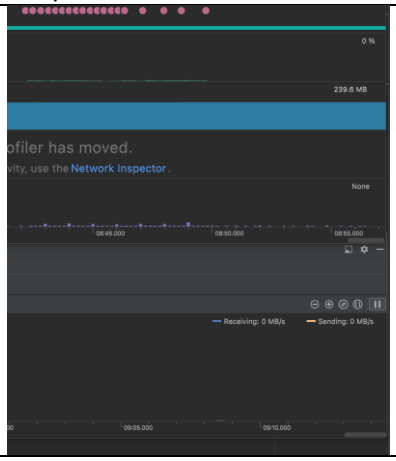
HU05 – Consultar listado de coleccionistas

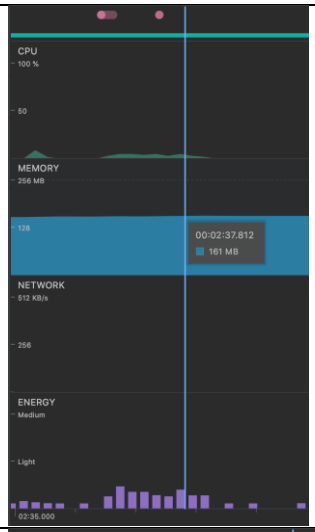
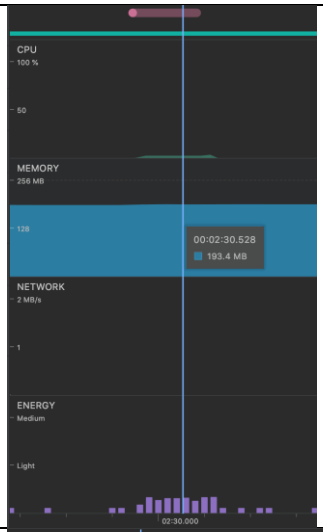
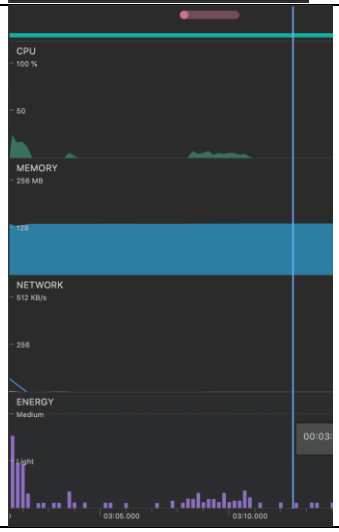
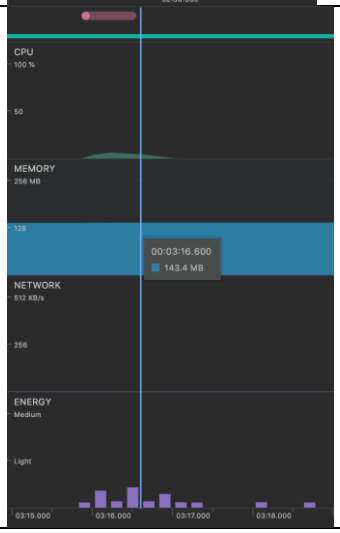
En el caso de consultar el catálogo de coleccionistas, se puede observar el mismo comportamiento que se evidenció en el listado de álbumes y artistas, después de las optimizaciones la cantidad de conexiones es más baja comparado con el antes, pero el uso de memoria incrementa.

Dispositivo	Antes	Después
1		

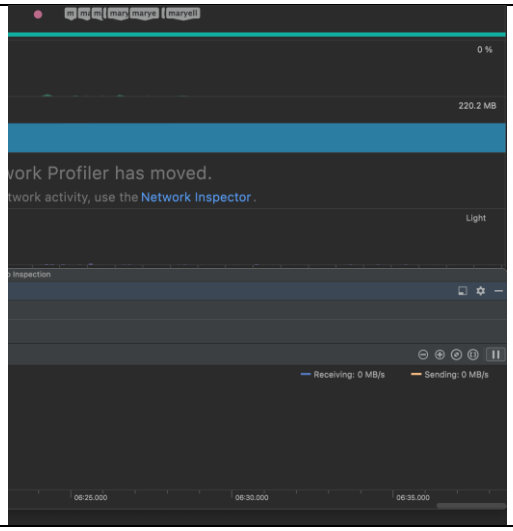
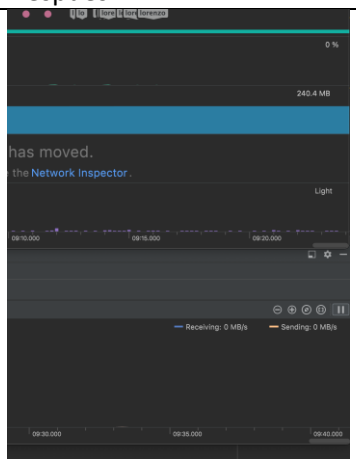
2		
3		

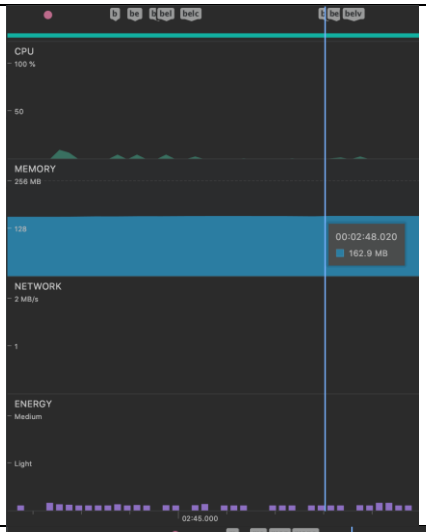
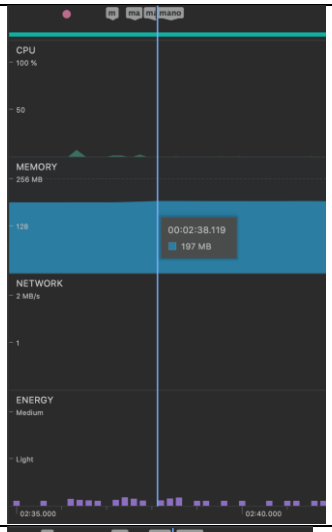
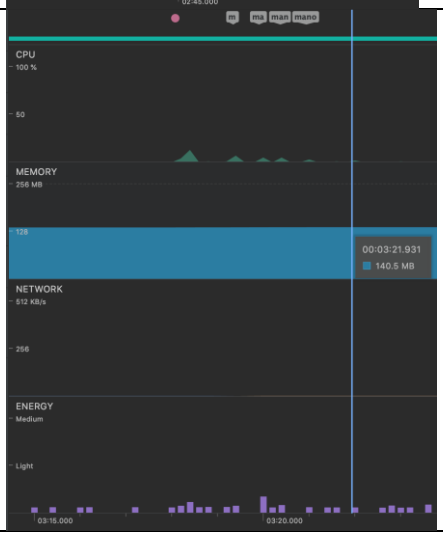
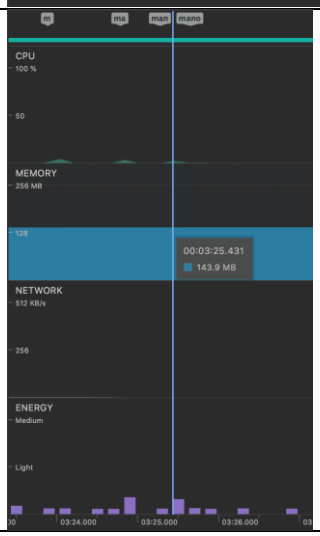
HU05 – Consultar listado de coleccionistas, haciendo scroll en el listado

Dispositivo	Antes	Después
1		

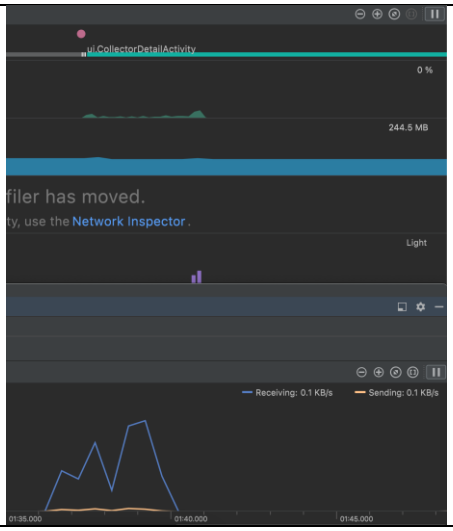
2		
3		

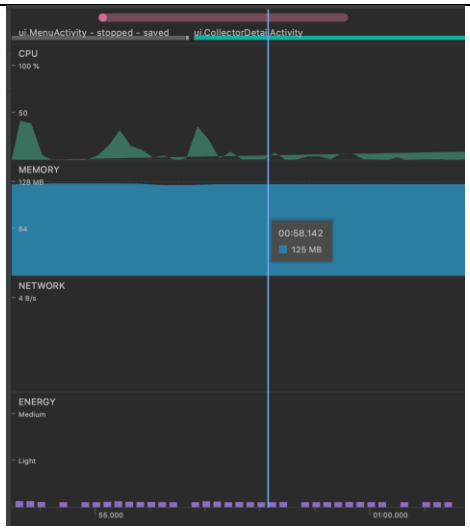
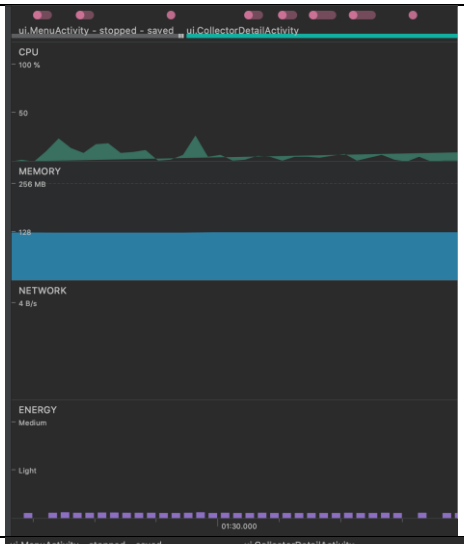
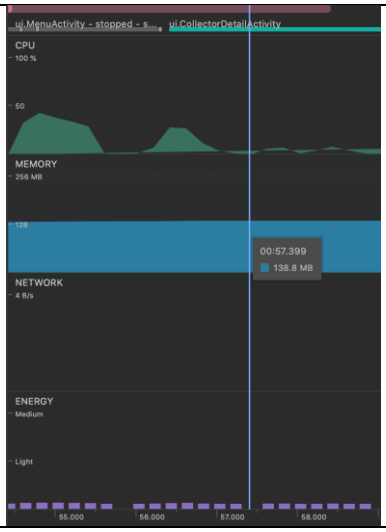
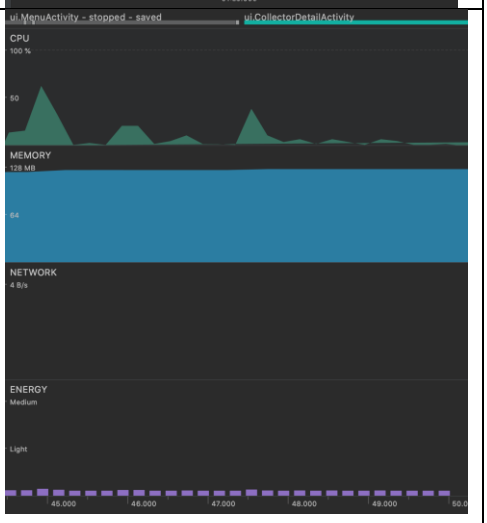
HU05 – Consultar listado de coleccionistas, filtrando por nombre

Dispositivo	Antes	Después
1		

2		
3		

HU06 – Consultar detalle de coleccionista

Dispositivo	Antes	Después
1		

2	 ui.MenuActivity - stopped - saved ui.CollectorDetailActivity CPU 100 % 50 MEMORY 128 MB 64 00:58.142 125 MB NETWORK 4 B/s ENERGY Medium Light 55.000 01:00.000	 ui.MenuActivity - stopped - saved ui.CollectorDetailActivity CPU 100 % 50 MEMORY 256 MB 128 NETWORK 4 B/s ENERGY Medium Light 01:30.000
3	 ui.MenuActivity - stopped - s... ui.CollectorDetailActivity CPU 100 % 50 MEMORY 256 MB 128 00:57.399 138.8 MB NETWORK 4 B/s ENERGY Medium Light 55.000 56.000 57.000 58.000	 ui.MenuActivity - stopped - saved ui.CollectorDetailActivity CPU 100 % 50 MEMORY 128 MB 64 NETWORK 4 B/s ENERGY Medium Light 45.000 46.000 47.000 48.000 49.000