VLC on Android Testsuite

Basic Information

Name: Shivansh Saini
Major: Computer Science and Engineering
Degree: Bachelor of Technology
Year: Sophomore
Institute: Indian Institute of Technology (BHU), Varanasi
Github: shivanshs9
IRC: TheFaker
Email Address: shivansh.saini.cse17@iitbhu.ac.in
Postal Address: A-1854 (Second floor), Greenfield Colony, Faridabad, Haryana, India - 121003
Phone number: (+91) 9717477205
CV: https://drive.google.com/file/d/1Jlza3wGcVI3QTbQhZ4FMk9CmHnsDVxQZ/view
Timezone: Indian Standard Time (UTC +5:30)

Project

Abstract

This project aims to contribute to the stability of the Android port of VLC, by writing test suites for the VLC user interface and the libVlc port for Android. Writing tests ensures that developers catch the regressions at an early stage and the deployed application stays stable.

Motivation

For every new piece of code written in any big project, like VLC on Android, the chances are that old things may break and new features may have some bugs. These could lead to instability of the otherwise working system, and hence detrimental for the software and its end users. Moreover, amidst the planned release cycles, it's vital for developers to move forward with their code without getting pushed back by bugs.
Hence, it's vital to write automated tests to:

- Test each component of the application or unit of code individually - called **"Unit Test"**.
- Test together multiple components at the same time - called **"Integration Test"**.
- Test the entire application, just as an end user would - called **"Functional Test"**.

Moreover, there's another benefit of writing functional tests which are usually underappreciated - they provide documentation of what the code is supposed to do. That's why, I believe that the **"Behavior-Driven Development"** would greatly help the new developers to work on their ideas

for the VLC application considering the rest of the code just like a **Black box**, that is being unaware of the internal workings of the rest of the application.

Why VideoLAN?

I am only applying for VideoLAN organisation for GSoC' 19 and have no plans either to invest in other organisations for this summer. The VLC media player was the first open source software I used, at the time when I didn't even know the meaning of open source. I got motivated by the journey VideoLAN has gone through from the VLC media player we all know and love. However, mostly, I got a drive to get involved in this organization when the president of VideoLAN, Jean-Baptiste Kempf, came to our institute during Technex 2018 edition.
His think talk was really inspiring for me and drove me to contribute to open source. Just as he said, "Keep doing it", I'll be contributing to the projects for VideoLAN as much as I can.

Proposed Deliverables

1. Write test suites around libVLC and medialibrary on Android, so that the core framework stays stable.
2. Write test suites for the application's interface to ensure the correct behaviours of UI views.
3. Write integrated tests around the main playback components.
4. Write functional tests to test the overall application, just like a user would.
5. Aim to maximise test coverage of libVlc and vlc-android.
6. Run the written tests at least once per day with the latest master code on the gitlab platform, so that no regressions miss.
7. Write documentation for the libVlc code, so that the new developers can understand the project faster.

Plan of Action

I believe that tests can very well explain the intention of the programmer for a piece of code, better than comments can do. That's why I plan to follow the so-called **Behavior-driven development (BDD)** approach for writing tests. In this approach, more focus is placed on testing behaviour, rather than implementation. When writing tests for **Black Box Testing**, the tester isn't aware of implementation details. In contrast, **White Box Testing** implies that the test is aware of the internal workings of the code.

I'll write **Unit Tests** for the core classes to test that their methods work as expected, regardless of their implementation. Since the focus is only on one element (our **System under Test** or

SUT) in isolation, its dependencies are faked, so our tests do not rely on them. So, these are **Test Doubles**; a pretend object used in place of a real object (Nice article).
For cases where we're just concerned with state verification, it's preferred to use **Stubs**, whereas, for behaviour verification, it's preferred to use **Mocks**. After understanding the VLC codebase, the best approach to write tests without many side-effects would be to use mocks, as expected from BDD approach. So, I'll be writing unit tests as a mockist BDD practitioner. I also got inspiration from this article.

Going up the **Testing Pyramid** (for Android), I'll write **Integration Tests** to test several classes together by checking how they behave. These include testing with the Android Framework as a Runtime environment (without the complexity of UI framework) and runs respectively for different SDK versions, to ensure the tests pass for all the supported Android versions. In these tests, test doubles would be awkward elements or those elements which we can't control.

Both Unit and Integrations Tests can be run very frequently on a CI server with every new commit. Hence they are perfect for catching regressions without slowing down the development time. These tests are also quite reliable because they aren't affected by side-effects from hardware or network issues.

Finally, **Functional Tests** are the most complex and can only be run once a day (Training article). These are very critical to test the common workflows and involve the complete stack, like network, UI and logic layers. Hence, these tests are highly unreliable and slow to execute.

Test Coverage is the key metric in today's competitive market for a quality product. Higher the percentage more likely is it for the software to be less prone to regressions. However, it's been proven many times that code coverage is not the best indicator for test strength and thoroughness. That's why I wish to later focus on **Mutation Testing** as the key index for the test strength (Nice article and Proof of Concept).
Due to time constraints, I may not be able to work on it during the summer. But, I wish to work on it even after the GSoC period is over.

Timeline

Community Bonding Period — May 6 - 27

I will discuss the project in further detail with the mentor. I will also get more acquainted with the codebase, especially the Playback controls and **VLCVideoLayout**. I'll try to fix some issues to get ideas on the priority of the classes to test. I will try and experiment more with the currently proposed idea to write tests, with inputs from the other developers. The optional parts, like Coveralls and fastlane, would especially be discussed with the mentor.

Week 1 — May 28 - June 3

This period would be spent on setting up the utility methods for creation of fake Media objects and to write the exact list of native method calls which would be stubbed with our test data.

Weeks 2 & 3 — June 4 -17

I'll write unit tests for the mentioned classes in LibVLC and MediaLibrary. While writing these, I'll stub the parts which require actual hardware support, like shared libraries and VLCObject events. So, in this period, I'll have completely filtered out the Android components and the native components.

Week 4 & 5 — June 18 - July 1

I'll write unit tests for the media, providers and viewmodels packages of the vlc-android app. By the end of this period, more than half of unit tests would be done. Also, I'll update the configuration file for Gitlab CI with a new job to run the written tests and to generate the artefact reports.

Week 6 & 7 — July 2 - 15

I'll wrap up the remaining unit tests and complete the integration tests. I'll also start experimenting the above-mentioned ways for automated running of instrumentation tests.

Week 8 & 9 — July 16 - 29

I'll also discuss with the mentor for the exact flows to automate for functional tests. I'll start recording the essential interactions with the app and test locally. Meanwhile, I'll also plan out with the mentor on how exactly the actual media files can be stored to be used in emulator for functional tests.

Week 10 & 11 — July 30 - August 13

I'll focus on finishing the instrumentation test scenarios with actual sample media files. Meanwhile, I'll work on a script to run multiple emulators for testing with sample media files pushed. I'll update the gitlab CI configuration file with different jobs for each device variant.

Ultimate Week — August 14 - 19

For the final week, I'll work on test coverage reports and cleanup of my final submission. I'll also analyse the written tests and their implementation in CI, and discuss with the mentor on what more things could've been done to make it better.

Detailed Plan and Implementation

Required Dependencies:

- JUnit4 (EPL-1.0 License)
- Mockito (MIT License)
- PowerMock (Apache-2.0 License)
- Robolectric (MIT License)
- kotlinx-coroutines-test (Apache-2.0 License)
- androidx.test APIs
- Espresso (Apache-2.0 License)
- livedata-testing (Apache-2.0 License)
- JaCoCo (EPL-1.0 License)
- JaCoCo gradle plugin

Optional Dependencies:

- fastlane (MIT License)
- automated-test-emulator-run (MIT License)
- Coveralls-gradle-plugin (MIT License)
- Coveralls (free for open source projects)

Unit tests

While it would be nice to switch to **JUnit5** for the Test Suites, it's currently impossible due to lack of official support in Android (Issue). So for now, I will write tests in **JUnit4** testing framework.
For test doubles, I'll go by mock approach and use **Mockito** as the Mocking framework for our unit tests. Its verification syntax is neat and so can be used to write readable tests. Moreover, using mocks should make testing multithreaded code simpler by mocking the concurrent jobs and writing tests synchronously. This ensures the reliability of the tests.
To enable mocking of static methods in some awkward situations, **PowerMock** is used. It allows easy access to internal state by manipulation of bytecode and custom classloader. Moreover, it has extensions over existing mocking frameworks, including Mockito, leading to similar syntax. This allows me enough flexibility to not change the design of rest of the code while writing good tests.
To test LiveData, **livedata-testing** library is used to model real code usage. It has a fluent API of assertions - a necessity when writing good tests. With it, I can write tests as pure JUnit tests leading to fast execution.
Unfortunately, checking the app codebase, there are lots of dependencies on Android as a library (like TextUtil, Uri). In these cases, we don't need to access Android as a Runtime environment, yet our unit tests fail because of the default stub implementation of Android framework (Google issue). Besides mocking every Android code, we could run our test on Robolectric test runner. However, that's a very slow approach because of setting up Application and Resources which are unneeded here.

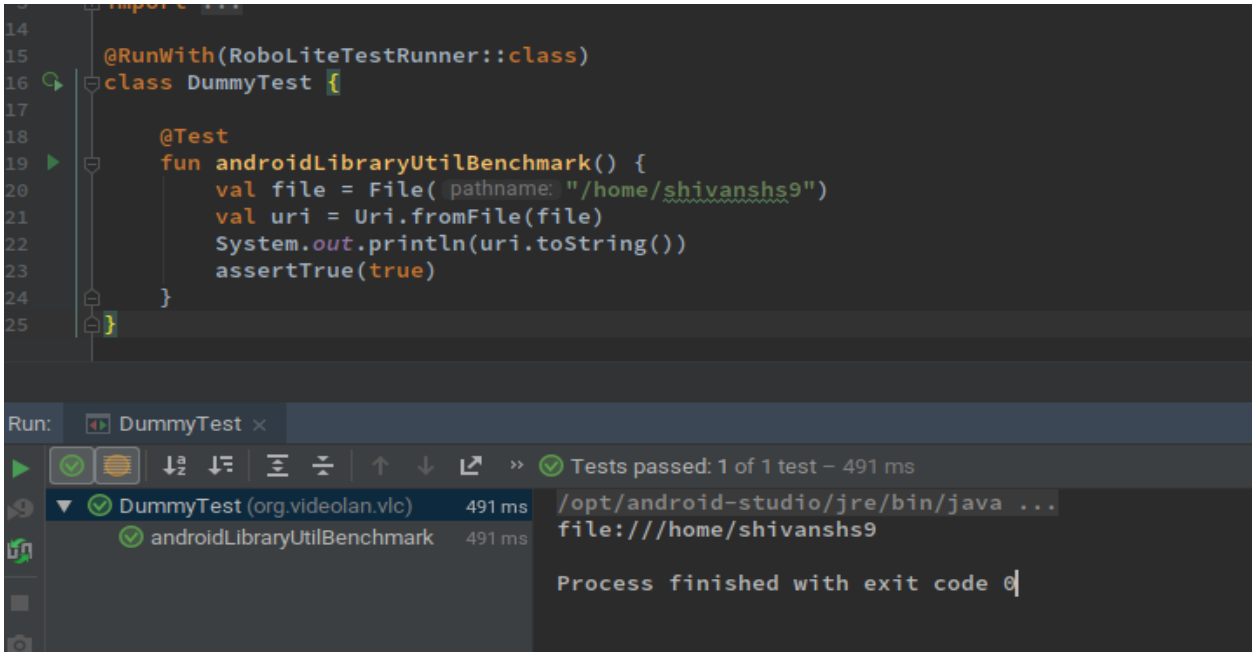
According to [this suggestion], I could use either [unmock-plugin] or a custom **RobolectricTestRunner** class. Testing both approaches, I did a small benchmark and had almost the same results:

With **RoboLiteTestRunner** class (**491ms**):

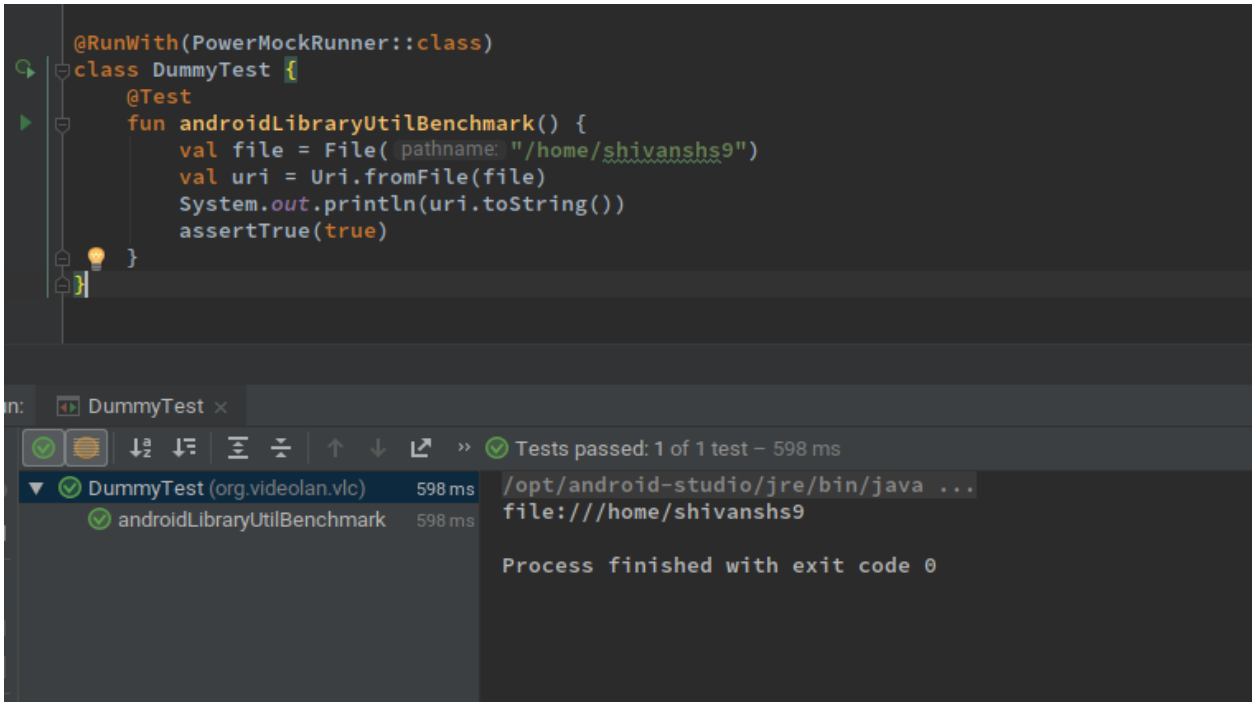
```
class RoboLiteTestRunner(cls: Class<*>): RobolectricTestRunner(cls) {
   override fun beforeTest(sandbox: Sandbox?, method: FrameworkMethod?,
bootstrappedMethod: Method?) {
   }

   override fun afterTest(method: FrameworkMethod?, bootstrappedMethod:
Method?) {
   }

   override fun getHelperTestRunner(bootstrappedTestClass: Class<*>?):
SandboxTestRunner.HelperTestRunner {
       try {
           return SandboxTestRunner.HelperTestRunner(bootstrappedTestClass)
       } catch (initializationError: InitializationError) {
           throw RuntimeException(initializationError)
       }
   }
}
```



With **unmock-plugin** (**598ms**):



Since they both took a similar time, I considered which one is more appropriate to use. A custom test runner seems more appropriate to me, because of lack of any new dependency and it's more explicit when it comes to use. The tests which require Android as a library can be explicitly run on **RoboLiteTestRunner**, rather than applying a Gradle plugin which applies to every test suite. It allows flexibility to change SDK config using Robolectric.
Considering Robolectric has strongly recommended using androidX test APIs (link), *build.gradle* will be changed:

```
dependencies {
    testImplementation "androidx.test:core:$rootProject.ext.androidxVersion
"
}
```

Also, to support PowerMock with our custom test runner, a new **PowerMockRule** will be added to our test suites (wiki).

Following are the classes for which I'll write Test Suites:

1. **org.videolan.libvlc.VLCObject -** Base class for all classes corresponding to C++ LibVlc VLCObject and handles locks & event listeners.
2. **org.videolan.libvlc.LibVLC -** A wrapper class over C++ LibVLC class to handle loading shared libraries.
3. **org.videolan.libvlc.Media -** Encapsulates a single Media information and all its related events.

4. **org.videolan.libvlc.MediaList -** Encapsulates an array of Medias and its addition/deletion events.
5. **org.videolan.libvlc.MediaDiscoverer -** Encapsulates discovery of MediaList through native calls.
6. **org.videolan.libvlc.MediaPlayer -** Encapsulates state of currently-playing media and its events.
7. **org.videolan.libvlc.util.MediaBrowser -** Start up media discoverers for different categories and allows to listen to media events.
8. **org.videolan.medialibrary.Medialibrary -** Encapsulates media categorizations, events and initialization of native libraries.
9. **org.videolan.medialibrary.media.MediaWrapper -** Encapsulates a single media information and metadata.
10. **org.videolan.medialibrary.media.Album -** Encapsulates a single album information.
11. **org.videolan.medialibrary.media.Artist -** Encapsulates a single artist information.
12. **org.videolan.medialibrary.media.DummyItem -** These are used for creating dummy media items.
13. **org.videolan.medialibrary.media.Folder -** Encapsulates a single folder information.
14. **org.videolan.medialibrary.media.Genre -** Encapsulates a single genre information.
15. **org.videolan.medialibrary.media.HistoryItem -** Encapsulates information for a media in history.
16. **org.videolan.medialibrary.media.Playlist -** Encapsulates a single playlist information.
17. **org.videolan.medialibrary.media.Storage -** Encapsulated a URI to represent media item as file.
18. **org.videolan.vlc.media.MediaGroup -** Encapsulates group of MediaWrappers as an object of MediaWrapper itself.
19. **org.videolan.vlc.media.MediaUtils -** Utility methods to add, play and open medias via **PlaybackService**,
20. **org.videolan.vlc.media.MediaWrapperList -** Encapsulates list of MediaWrappers to be used by **PlaylistManager**.
21. **org.videolan.vlc.media.PlayerController -** Provides actions to affect behaviour of **MediaPlayer** depending on preferences, and utility methods for playback state of the current media.
22. **org.videolan.vlc.media.PlaylistManager -** Created by **PlaybackService**, this class manages the current **MediaWrapperList** and also uses **PlayerController** to affect current playback.
23. **org.videolan.vlc.providers.FileBrowserProvider -** Media data provider via livedata and coroutine channels. Also converts **Media** object to **MediaWrapper** object.
24. **org.videolan.vlc.providers.FilePickerProvider -** For this provider, favorites can't be shown and is used to browse storage files.
25. **org.videolan.vlc.providers.NetworkProvider -** Browse through files via LAN.
26. **org.videolan.vlc.providers.StorageProvider -** Used to provide storage directory to choose for medialibrary to scan.

27. **org.videolan.vlc.viewmodels.SortableModel -** ViewModel abstract class to provide basic sorting functionality.
28. **org.videolan.vlc.viewmodels.BaseModel -** Base class to provide **MediaLibraryItem** as a **LiveDataset**.
29. **org.videolan.vlc.viewmodels.HistoryModel -** ViewModel class to provide recently played media files.
30. **org.videolan.vlc.viewmodels.MediaLibraryModel -** Base class which implements methods for medialibrary lifecycle to refresh data.
31. **org.videolan.vlc.viewmodels.PlaylistModel -** Encapsulates the lifecycle of **PlaybackService** using LiveData and also keeps progress and playerState in LiveData.
32. **org.videolan.vlc.viewmodels.StreamsModel -** ViewModel class that provides last played network streams data.
33. **org.videolan.vlc.viewmodels.VideosModel -** ViewModel class that provides list of videos from medialibrary.
34. **org.videolan.vlc.viewmodels.SubtitlesModel -** ViewModel class to search and download subtitles.
35. **org.videolan.vlc.viewmodels.audio.AlbumModel -** Used by **AlbumsFragment** for TV media browser.
36. **org.videolan.vlc.viewmodels.audio.ArtistModel -** Used by **ArtistsFragment** for TV media browser.
37. **org.videolan.vlc.viewmodels.audio.GenreModel -** Used by **GenresFragment** for TV media browser.
38. **org.videolan.vlc.viewmodels.audio.TracksModel -** Used by **TracksFragment** for TV media browser.
39. **org.videolan.vlc.viewmodels.browser.BrowserModel -** ViewModel class that communicates with respective **BrowserProvider** implementation, using LiveDataset.
40. **org.videolan.vlc.viewmodels.paged.PagedAlbumsModel -** ViewModel class that allows paging on albums data and sorting.
41. **org.videolan.vlc.viewmodels.paged.PagedArtistsModel -** ViewModel class that allows paging on artists data.
42. **org.videolan.vlc.viewmodels.paged.PagedFoldersModel -** ViewModel class that allows paging on folders data.
43. **org.videolan.vlc.viewmodels.paged.PagedGenresModel -** ViewModel class that allows paging on genres data and sorting.
44. **org.videolan.vlc.viewmodels.paged.PagedPlaylistsModel -** ViewModel class that allows paging on playlists data.
45. **org.videolan.vlc.viewmodels.paged.PagedTracksModel -** ViewModel class that allows paging on medias data and sorting.
46. **org.videolan.vlc.util.Browserutils -** Utility class user by implementations of **BrowserProvider**.
47. **org.videolan.vlc.util.FileUtils -** Utility class that provides methods to get, copy, delete files.

48. **org.videolan.vlc.util.FilterDelegate -** Utility class that provides filtering implementation for **MediaLibraryItem** and in **Playlist**.
49. **org.videolan.vlc.util.KExtensions -** Various Kotlin extensions related with MediaLibrary and other utilities.
50. **org.videolan.vlc.util.LiveDataset -** LiveData of a mutable list of a generic class.
51. **org.videolan.vlc.util.LiveDataMap -** LiveData of a mutable map of a generic class.
52. **org.videolan.vlc.util.ModelsHelper -** Provides utility methods for a collection of **MediaLibraryItem** to split and generate sections with headers, depending on sort mode.
53. **org.videolan.vlc.util.MurmurHash -** A class to provide implementation of Murmur Hash, a fast-hash function.
54. **org.videolan.vlc.util.PathUtils -** Provides utility extensions over strings related to file path.
55. **org.videolan.vlc.util.Permissions -** Provides methods to get the current state of permissions and helper method to create dialog to ask for such permissions.
56. **org.videolan.vlc.util.Preferences -** Provide static methods to get and store floats array in Shared Preferences.
57. **org.videolan.vlc.util.RendererLiveData -** Wrapper class over LiveData of **RendererItem** to ensure the both new and old RendererItem objects are properly retained and released.
58. **org.videolan.vlc.util.Strings -** Provides kotlin extensions for easier conversion between strings and other type, like number.
59. **org.videolan.vlc.util.ThumbnailsProvider -** Utility class that provides thumbnails and also composes images for use with folder and group.
60. **org.videolan.vlc.util.Util -** Provides static utility methods of miscellaneous category.
61. **org.videolan.vlc.util.VLCAudioFocusHelper -** Utility class to request audio focus from android.
62. **org.videolan.vlc.util.VLCDownloadManager -** A receiver class to handle download completion and new downloads.
63. **org.videolan.vlc.util.VoiceSearchParams -** A utility class that parses the query and extras bundle passed from **MediaSessionCallback** to handle music search.
64. **org.videolan.vlc.ExternalMonitor -** A receiver class that listens to network, storage and OTG events, and correspondingly updates **MediaLibrary** devices.
65. **org.videolan.vlc.MediaParsingService -** Used to start MediaLibrary and for scanning devices with it. Also, shows a scan notification and updates the scan progress.
66. **org.videolan.vlc.MediaSessionCallback -** Implements the callback for Android' MediaSession for playback methods.
67. **org.videolan.vlc.PlaybackService -** Sets up a receiver for remote playback actions and performs appropriate action on **PlaylistManager** depending on desired state. It manages notification for the playing media. It is also responsible for handling media events and running callbacks.
68. **org.videolan.vlc.RecommendationsService -** Notifies the user with media recommendations based on last 3 recently played medias.

While implementing the tests, I will also have to override the **Dispatchers.Main** coroutine context so that our tests can make do without using Android Main Looper. For that, I'll include in app's *build.gradle*:

```
dependencies {
    testImplementation
"org.jetbrains.kotlinx:kotlinx-coroutines-test:$rootProject.ext.kotlinx_ver
sion"
}
```

Then, before running the test:

```
   @Before
    fun setUp() {
        Dispatchers.setMain(Dispatchers.Unconfined)
    }
```

As an example, here's a sample code for testing **StreamsModel** where I've mocked native method, ***nativeInit*** of **MediaLibrary**, and used **livedata-testing** for assertions in livedata:

```
@Test
fun failedInitialization_GetEmptyCollection() {
   PowerMockito.doReturn(Medialibrary.ML_INIT_FAILED).`when`(mockedLibrary,
"nativeInit", any(), any())

   mockedLibrary.init(mockedContext)

   streamsModel.updateHistory()
   streamsModel.observableHistory
           .test()
           .awaitValue()
           .assertHasValue()
           .assertValue(Medialibrary.EMPTY_COLLECTION)
}
```

Integration tests

For integration tests, the complete android framework is mocked in runtime environment, which will allow to load views, resources etc, in the test suite. These will ensure that our application code performs well with external services. These tests also include testing multiple classes in a single test suite, so that their behaviour can be tested as a whole.
Following are the core classes which will be tested, along with their immediate dependencies:

1. **org.videolan.vlc.util.AndroidDevices -** Utility methods for android features and standard values for Media directories. Dependency on Android framework for many of the core features.
2. **org.videolan.vlc.util.HttpImageLoader -** Utility class to download image and cache it.

3. **org.videolan.vlc.gui.audio.AudioAlbumsSongsFragment -** Fragment to browse by albums and songs.
4. **org.videolan.vlc.gui.audio.AudioBrowserFragment -** Can browse by Artists, Tracks, Albums or Genres when navigating audio.
5. **Media, MediaList, MediaDiscoverer and MediaBrowser -** These interact with each other to keep track of media changes and the current list.
6. **PlaylistManager, PlayerController and PlaybackService -** These interact with each other for media playback, state and information.

The main hurdle is to test any UI which requires drawing video on a surface view. Those can only be performed on actual hardware because of dependency on required CPU architecture. Also, since there is no actual storage option, sample media files will have to be added temporarily. That's why I propose to mock the native calls which are responsible for actual parsing of media files to return pre-created **MediaLibraryItem** objects. A test utility class is responsible for creating these objects and a sample file is created for each object in the corresponding public directories. For example, for music directory I can use:

```
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_MUSIC).
absoluteFile
```

Functional tests

These tests are supposed to test a particular flow on an emulated Android device or actual device. The User Interactions within the app will be tested, by using Espresso testing framework. These tests will use **androidx.test.runner.AndroidJUnitRunner** instrumentation runner, as mentioned in the official docs, so that the test package can be run together with the application package under the same process by the Instrumentation component of Android framework. These tests will run on a separate **Instrumentation thread** and only run when the main UI thread is idle to avoid inconsistency with tests' results. For ease of writing tests, I'll use Espresso Test Recorder to record the flow to test. Since these tests also requires actual media files, sample files have to be pushed beforehand to the emulator (using **adb push**).

The main flows to test are:

- Interaction with playback controls when playing audio and video.
- Interaction with media browser - selection of media files, options etc.
- Interaction with preferences screen.
- Interaction with the side navigation in **MainActivity**.

Since these tests are slow, these are run only once per day on Gitlab CI/CD using pipeline schedules. To run these instrumented tests, multiple AVDs have to created and run with different specifications to cover multiple API variants. Moreover, these steps have to be automated, and the emulators have to be headless to run in a non-graphical CI server. For this, we need to install the required android system images so that AVDs can use them. Currently, only the **android-26** target is installed.

Dockerfile for vlc-debian-android (L50):

```
    tools/bin/sdkmanager "build-tools;26.0.1" "platform-tools"
"platforms;android-26" && \
```

Once that's done, new AVDs will have to be created by using *tools/bin/avdmanager* and mention the target ID. Once created, emulators will have to be started and the tests will run on them:

```
instrumentation_tests:
  stage: test
  script:
    - emulator -avd testAVD -no-audio -no-window & # To create an headless
emulator with no audio support.
    - ./ci/android-wait-for-emulator.sh # Script to wait for emulator to fully
boot.
    - adb devices
    - adb shell input keyevent 82 & # To unlock the booted android.
    - ./gradlew connectedAndroidTest # To run the actual instrumentation tests.
    - ./ci/stop-emulators.sh # Script to stop the running emulator.
```

For multiple virtual devices, it's possible to write them as separate jobs in the same stage so that they do run in parallel (docs). However, it'll still be slow due to the setup of the environment for each job. To avoid that, we could use fastlane and its plugin, automated-test-emulator-run. Using these we could simply mention the AVD config in a JSON file and use that in a single lane or multiple ones with sharding (article). In this article, fastlane is used for Gilab CI in android project.

Test Cycle & Coverage

The written unit tests will be run in **Gitlab CI/CD** Pipeline after every new Merge Request or a commit in Master branch. This ensure that any change does pass the test and any regressions are avoided. Gitlab CI/CD already powers the project right now, and it's a wonderful built-in solution.

JaCoCo tool is a free Java library to evaluate code coverage depending on project requirements. Following this guide and the JaCoCo Gradle Plugin docs, I would have to change the project's *build.gradle* and *.gitlab-ci.yml*. This guide proved helpful to customise the JaCoCo's behaviour for higher flexibility.

build.gradle

```
buildscript {
   dependencies {
       classpath "org.kt3k.gradle.plugin:coveralls-gradle-plugin:2.6.3"
   }
}
```

.gitlab-ci.yml

```yaml
continuous-app-build:
    extends: .build-all-base
    script:
        - ./compile.sh --init
        - ./gradlew assembleDebug
    only:
        changes:
            - vlc-android/**/*
            - assets/**/*
            - .gitlab-ci.yml
    except:
        - schedules

### Run app unit tests.
continuous-debug-test:
    stage: test
    script:
        - ./gradlew testDebug

### Run app coverage tests. Requires secret token for coveralls.
continuous-coverage-test:
    stage: test
    script:
        - ./gradlew jacocoTestReport coveralls
    variables:
        COVERALLS_REPO_TOKEN: ### to be provided.
```

Coveralls is an optional part of the project since gitlab do provide the useful ability to generate artefact reports from JUnit (Proof of Concept). However, I find it to be a highly motivating factor for new developers to try and write test suites for their added feature, to increase that coverage percentage. Moreover, this also promotes open reports for others to review.

Personal Information

Personal Details

I'm Shivansh Saini, an undergraduate at Indian Institute of Technology (BHU), Varanasi. I've been fascinated with software and games development since I was 14. I mainly started with coding 2 years back and have grown to like open source, because of its vibrant and engaging community. I actively use open source softwares and like to contribute back when I can. I like to hack around with stuff and learn new things, such as fixing Hackintosh or studying about Wayland. I tend to place more importance on practical implementations rather than theoretical study. I believe in solving problems by reusing the latest technological stuff, rather than reinventing the wheel. I've also been working in a startup for almost a year now and so have come to appreciate the power of teamwork.
I'm proficient in C, C++, Python, Java, Javascript and Shell scripting. I've basic knowledge of Kotlin, but insufficient experience in it. I have experience in web backend development using

MVC frameworks such as Django (usually REST APIs) and frontend development using React and VueJs. I have experience in Android development of 9 months, using the MVVM architecture. I am an eager learner and ready to add more skills to my toolbox.
I use Manjaro Linux, with zsh as my favoured shell. I use Sublime Text for small codes and Visual Code (Code-OSS) for working on development projects.

Communication

I'm flexible with my schedule and have inculcated the habit of working at night, so time zone difference shouldn't be an issue. I'm comfortable with any of the communication mediums I mentioned above. However, due to network issues, I may not receive messages via IRC.

I can work full-time on weekdays and am usually available between 12 PM IST to 3 AM IST. On weekends, I would love to spend time communicating with the team to learn from them, while working on whatever issues occur at that time. When the time permits, I would also like to contribute to the documentation of the code.

I may be travelling for a few days, and I'll inform those plans beforehand to my mentor. I'll also responsibly keep my mentor updated in case of any emergency that occurs with suitable details.

Post GSoC

If there are things left unimplemented, I'll try to complete them post GSoC and will keep contributing to VLC Android. I would like to contribute more to documentation of the code and to work on other things, like coverage reports and mutation testing.

Contributions

Issues opened:

- GSoC project: VLC on Android Testsuite - A task issue to update with the list of planned test suites.

Merge Requests assignee:

- Close button in notification to stop unpausable media. - Adds a close button in notification for unpausable medias (Fixes #181)
- WIP: Simple test suite for StreamsModel. - Has a single test suite for **StreamsModel** class.
- WIP: Allows VLC play in PiP mode from other apps. - Adds the ability to play videos in PiP mode from other apps.