GSoC 2021 Proposal

Introduce Support for Displaying Copyright Licenses (Abhay Garg)

About You

Why are you interested in working with Oppia, and on your chosen project?

Oppia's mission is to provide free education to those children who are unable to attend school for any reason or have to leave their school in between and I believe that this cause is very noble and it is a great step to make basic education accessible all over the world and hence, it is one of the most important reasons why I want to contribute to Oppia.
Also, I like the encouraging environment for beginner-contributors at Oppia and I learned a lot of new things while contributing to Oppia. I came to know about the best practices to code in general and also while implementing a new feature, fixing an existing bug, and writing test cases.
Achieving all the requirements of this project in the most efficient and scalable way will make me learn many new things, and this experience will also help me in the future as the feature that will be implemented through this project is a very prevalent (and essential) feature for any app, so the experience that I will gain through this project will also be very beneficial for my future career as a Software Developer.

Prior experience

I have been learning Android Development for one year and I have been contributing to open-source for about 9 months. In the initial days of my learning, I tried making an Android app for the infamous Pictionary game ***Skribbl*** as me and my friends loved this game and we played it a lot during the quarantine. We usually used to play it on our phones, and my friends complained that the drawing board is quite small in it and the UI also doesn't look very good on phones, so I decided to make an app for the game myself. I named it Sckribbel and used Firebase (as backend) to maintain the scores and handle the timer for drawing.
I am a member of the DSC-IIITL (Developer Students Clubs are powered by Google to promote Google Developer Technologies among University Students) and I recently conducted the *Android Study Jams Sessions* at my college to promote Android Development among students who are new to programming. In the first session, I taught some basics of Kotlin Language and the second session was a hands-on session, so I made a very basic and simple app in the session and introduced the students to the basics of Android App Development. These sessions were planned for the absolute beginners and thus involved only very basic stuff.

I also participated in SLoP(Semester Long Project) with Oppia as my mentor organization and I managed to get an overall rank - 18 with maximum points (350) amongst all the participants mentored by Oppia (Web and Android combined).
After I came to know about open-source and how it can be helpful to grow your learning and also provide you the experience to work on real-world projects, I started making small contributions to various Android Projects and my PRs got merged into 2 projects - Kiwix-Android and Oppia-Android. I liked the culture at Oppia and I found a lot of good-first issues here, so I continued contributing to the Oppia-Android project.
Gradually, I started feeling a bit confident and I started working on slightly tougher issues. While working on these issues, I came across very new things and I learned the importance of test-driven development and why writing good and correct test cases are necessary for any project as I saw breaking some old implementation while introducing a new feature or fixing an existing bug. This kept encouraging me to contribute more to the project as I was continuously learning something new.
Particularly, this project involves the implementation of a feature that will be directly visible to the users of the app and this is one of the most important reasons that this project interests me. Although the feature may look very simple, the actual implementation is not very easy as this has to be implemented in a way that is also extensible in the future. So, this project is challenging too which is another reason why I want to work on this project as it is always exciting for me to do something a bit challenging.

5 Best Merged PRs in Oppia Android -

1. #2482 - Fixed all the flaky test cases for OnboardingFragmentTest on Robolectric and Espresso and migrated OnboardingViewpager from ViewPager to ViewPager2.
2. #2671 - Fixed NavigationDrawer to highlight the correct menuItem on canceling the Switch Profile Dialog and on pressing the back button from various other activities like HelpActivity, OptionsActivity, etc.
3. #1923 - Fixed TickIcon disappearing bug in case of a long answer.
4. #2396 - Fixed long username's cutoff in HomeActivity.
5. #2351 - Migrated the ProfileInputView to TextInputLayout for AdminPinActivity.

Merged PRs in Oppia Web -

1. #11619 - Fixed end-to-end tests for ExplorationEditorImprovementsTab.js.
2. #11759 - Fixed end-to-end tests for GetStartedPage.js.

Contact info and timezone(s)

- E-mail - gargabhay2000@gmail.com
- Github - https://github.com/prayutsu
- Country - India
- LinkedIn - https://www.linkedin.com/in/prayutsu/
- University - Indian Institute of Information Technology, Lucknow

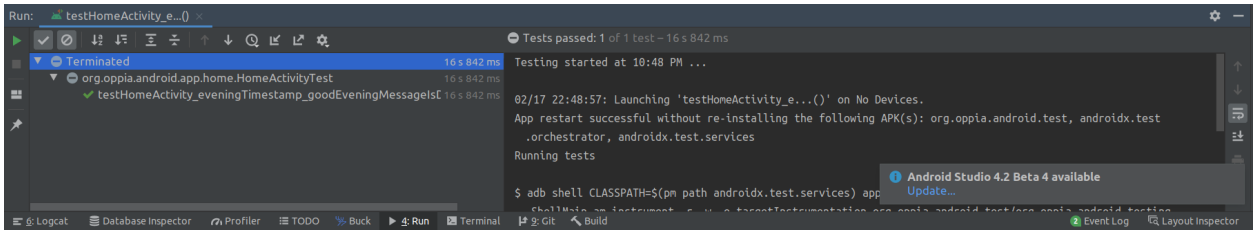
- Preferred Communication Method - Hangouts, e-mail
- Timezone - Indian Standard Time (IST)

Time commitment

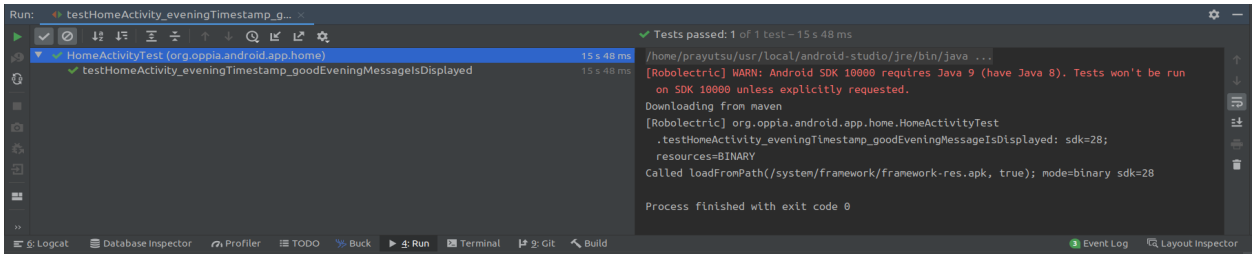
I will work on the project from Monday to Friday for 4-6 hours and take a rest on Saturdays and Sundays. If in case I am unable to work on a regular weekday, I will adjust this by working on weekends. So, I will be able to work 20-25 hours every week during the entire coding period.

Essential Prerequisites

- Yes, I can run a single instrumented Espresso test on my machine.



- Yes, I can run a single unit test on my local machine.



Other summer obligations

I don't have any specific commitments for this summer, and my exams will also finish during the Community Bonding Period only, so I will be fully devoted to this project.

Communication channels

I plan to communicate at least 2-3 times a week with my mentor and my preferred mode is Hangouts. However, I am also comfortable in communicating through email, or Gitter whichever suits us the most.
Mentors can generally expect a reply from me within a few minutes.

Application to multiple Orgs

No, I am only applying to Oppia.

Project Details

Product Design

The users of this feature are the Oppia Android app users and developers who would be able to see the list of libraries/dependencies and their corresponding Copyright Licenses on which the Oppia-Android app depends.

What is a Third-party Library?

Let us first understand what is a third-party library/dependency? A library is a collection of several code snippets that together simplify the implementation of a particular feature. It is designed to abstract away the exact logic to implement the feature and simplify the work of other developers. A library can be open-source or it can be proprietary that depends on the developer(s) of the library. Using different libraries to implement different features in software is similar to the manufacturing of any product, e.g, a car manufacturing company requires various components to build a car like wheels, good quality seat covers, gears, different types of oils, car engine, etc, so the company has two options - either to manufacture these different parts on its own or it could directly buy these parts from other companies that are already manufacturing these products with the best quality. In most cases, the second option fits the best because the car manufacturing company may only have the specialized engineers that are best in assembling the car optimally or enhancing the overall performance of the car, but while choosing the second option, the company ensures that it is using the best quality parts for its final product and it does not compromise the quality in any aspect. Similarly, developers use different libraries to implement complex features as it may not be possible for them to implement a particular feature in a better or optimized way than a particular library. It reduces the amount of code and also makes the main code of the software easier to understand.

Why attribute these libraries to the Oppia-Android App?

The Oppia-Android codebase uses various third-party libraries to implement various features easily and efficiently. Despite the code of most of these libraries is publicly available and is open-source, we want to credit the libraries that we use to provide the app and ensure the terms of their licenses are met.

User-journey of the finished project

After completion of this project, we'll have the following changes that will be visible in the actual app -

1. We'll have a dedicated UI in our app to display the list of the Maven dependencies of our app.
2. The list of third-party libraries would be accessible via the Help menu (Home Screen -> Main Menu -> Help -> Third-party Dependencies/Libraries).
3. The items of this list would display the dependency name and its version used in the app, and clicking on an item would take the user to a different screen and the Copyright License text of the corresponding dependency will be shown.

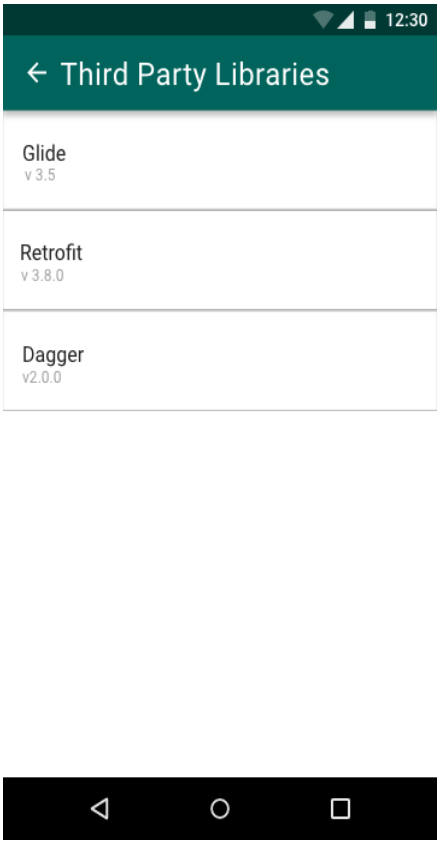
The list of dependencies and the license text itself would be scrollable as they would be large enough to not fit in a normal-sized phone screen e.g, Android phones have a screen size of about 6-7 inches these days.
Some of the dependencies for which the license text would be extracted are listed below. Please note that this list is not complete and the actual list would include more dependencies.

```
MAVEN_PRODUCTION_DEPENDENCY_VERSIONS = {
    "androidx.annotation:annotation": "1.1.0",
    "androidx.appcompat:appcompat": "1.0.2",
    "androidx.constraintlayout:constraintlayout": "1.1.3",
    "androidx.core:core": "1.0.1",
    "androidx.core:core-ktx": "1.0.1",
    "androidx.databinding:databinding-adapters": "3.4.2",
    "androidx.databinding:databinding-common": "3.4.2",
    "androidx.databinding:databinding-compiler": "3.4.2",
    "androidx.databinding:databinding-runtime": "3.4.2",
    "androidx.drawerlayout:drawerlayout": "1.1.0",
    "androidx.lifecycle:lifecycle-extensions": "2.2.0",
    "androidx.lifecycle:lifecycle-livedata-core": "2.2.0",
    "androidx.lifecycle:lifecycle-livedata-ktx": "2.2.0",
    "androidx.lifecycle:lifecycle-viewmodel-ktx": "2.2.0",
    "androidx.multidex:multidex": "2.0.1",
    "androidx.multidex:multidex-instrumentation": "2.0.0",
    "androidx.navigation:navigation-fragment": "2.0.0",
    "androidx.navigation:navigation-fragment-ktx": "2.0.0",
    "androidx.navigation:navigation-ui": "2.0.0",
    "androidx.navigation:navigation-ui-ktx": "2.0.0",
    "androidx.recyclerview:recyclerview": "1.0.0",
    "androidx.room:room-runtime": "2.2.5",
    "androidx.viewpager2:viewpager2": "1.0.0",
    "androidx.viewpager:viewpager": "1.0.0",
    "androidx.work:work-runtime": "2.4.0",
```

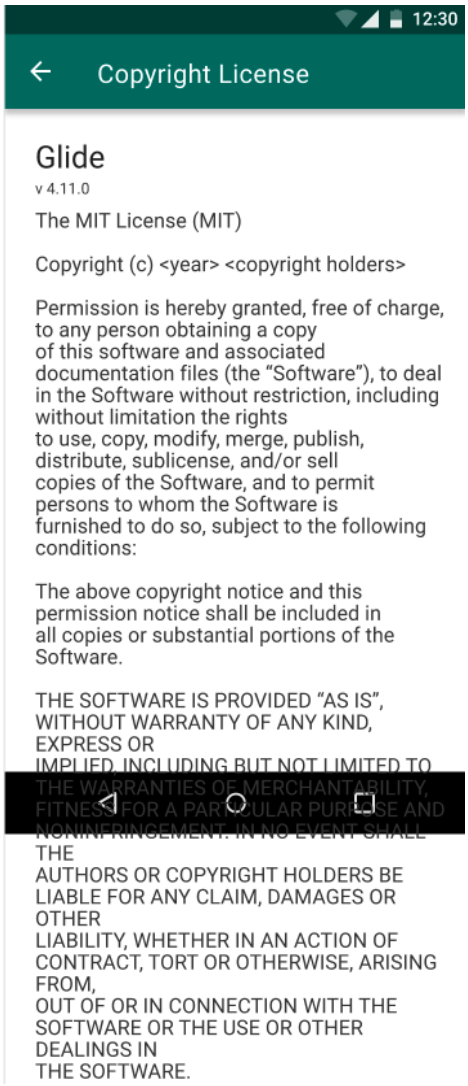
```
  "androidx.work:work-runtime-ktx": "2.4.0",
  "com.android.support:support-annotations": "28.0.0",
  "com.caverock:androidsvg-aar": "1.4",
  "com.chaos.view:pinview": "1.4.4",
  "com.crashlytics.sdk.android:crashlytics": "2.9.8",
  "com.github.bumptech.glide:glide": "4.11.0",
  "com.google.android.material:material": "1.2.0-alpha02",
  "com.google.android:flexbox": "2.0.1",
  "com.google.firebase:firebase-analytics": "17.5.0",
  "com.google.firebase:firebase-crashlytics": "17.1.1",
  "com.google.gms:google-services": "4.3.3",
  "com.google.guava:guava": "28.1-android",
  "com.google.protobuf:protobuf-lite": "3.0.0",
  "com.squareup.moshi:moshi-kotlin": "1.11.0",
  "com.squareup.moshi:moshi-kotlin-codegen": "1.11.0",
  "com.squareup.okhttp3:okhttp": "4.1.0",
  "com.squareup.retrofit2:converter-moshi": "2.5.0",
  "com.squareup.retrofit2:retrofit": "2.9.0",
  "de.hdodenhof:circleimageview": "3.0.1",
  "io.fabric.sdk.android:fabric": "1.4.7",
  "javax.annotation:javax.annotation-api:jar": "1.3.2",
  "javax.inject:javax.inject": "1",
  "nl.dionsegijn:konfetti": "1.2.5",
  "org.jetbrains.kotlin:kotlin-stdlib-jdk7:jar": "1.3.72",
  "org.jetbrains.kotlinx:kotlinx-coroutines-android": "1.3.2",
  "org.jetbrains.kotlinx:kotlinx-coroutines-core": "1.2.1",
}
```

The List of all the dependencies and the corresponding Copyright License texts would look similar to the below mocks -

List of All Dependencies



Copyright License Text



[Adobe XD Mock for Dependencies List Screen]

[Adobe XD Mock for License Text Screen]

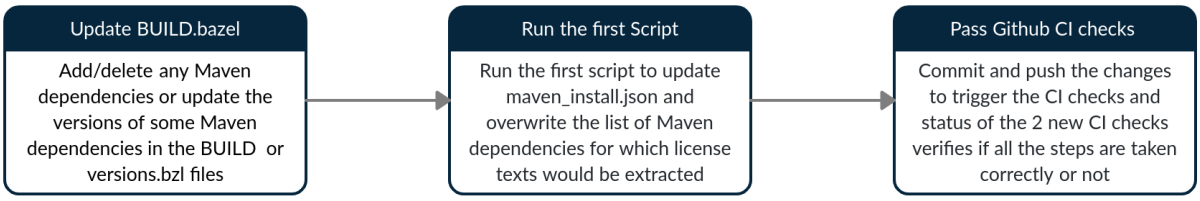
The project aims to achieve the above-mentioned functionalities by introducing 2 Kotlin scripts that would compile the list of all the Maven third-party dependencies along with their versions and license links, and also generate the XML string resource files containing dependencies' - names, versions, license types, and license texts.
We will also introduce proper Github CI checks that would get triggered for every commit on a PR that is to be merged to the `develop` branch of the Oppia-android repository to ensure that the feature doesn't break at any point in time and it also remains scalable when new dependencies are added/deleted.

Steps to add/delete dependencies for a general contributor -

This project will also bring changes in the process of updating Maven dependencies in the `BUILD or versions.bzl` files. Contributors will then have to follow the below steps (if they add/delete or change some Maven dependency in their PR) to ensure that they do not fail the Github CI checks -

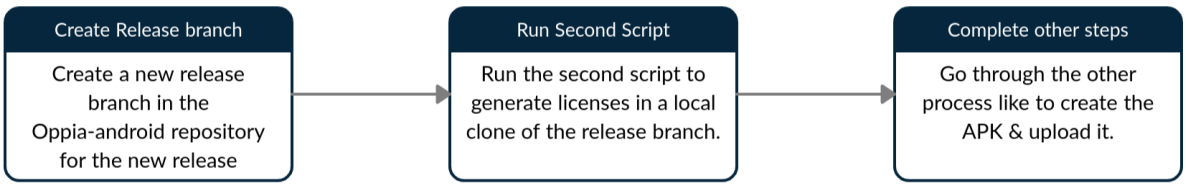
1. Update the dependency's artifact or version (or both) in the `BUILD` or `versions.bzl` files.
2. Run the first Kotlin script on their local fork. This script will first internally run the command `$ bazel run @unpinned_maven//:pin` so that the changes in the dependencies' artifacts or versions get reflected in the `maven_install.json` file and then recompile the list of dependencies.
3. Commit and push the changes and create/update the PR. If everything is done correctly, the 2 new Github CI checks (that will be added during the implementation of this project) will pass and if they fail, take a look at the reason for failure and repeat the above steps in the correct order again.

The below diagram helps us understand the flow of the Dependency Lifecycle -



Steps to Generate the Actual License Text Resource files -

1. Create a release branch for the new release.
2. Run the second script to generate licenses in a local clone of the release branch.
3. Go through our process to create the APK & upload it.



Technical Design

Architectural Overview

app module

The Oppia-android codebase follows a combination of MVP and MVVM architecture, so the new UI components will be introduced accordingly. For every new activity and fragment, a presenter will be introduced that will contain all the logic of the activity/fragment. ViewModels will be used to provide the data (i.e dependencies' names, versions, license texts, etc) to the activities/fragments.

The project involves minor changes in the below-mentioned files to accommodate the "Third-party Libraries" as a new option in the list of items of the HelpFragment's RecyclerView -

1. app/src/main/java/org/oppia/android/app/help/
    - HelpItems
        - A new option "THIRD_PARTY" needs to be added to the enum HelpItems.
    - HelpListViewModel
        - Currently, HelpListViewModel supports displaying only one item - "FAQ" in the helpFragmentRecyclerView so it will be modified to also support displaying the new option (Third-party Libraries).
    - HelpItemViewModel
        - Currently, HelpItemViewModel only knows what to do when the user clicks on the "FAQ" item of the helpFragmentRecyclerView and it needs to be extended to support the new option (Third-party Libraries) also.

The following components will be added to the app module of the repository along the specified path -

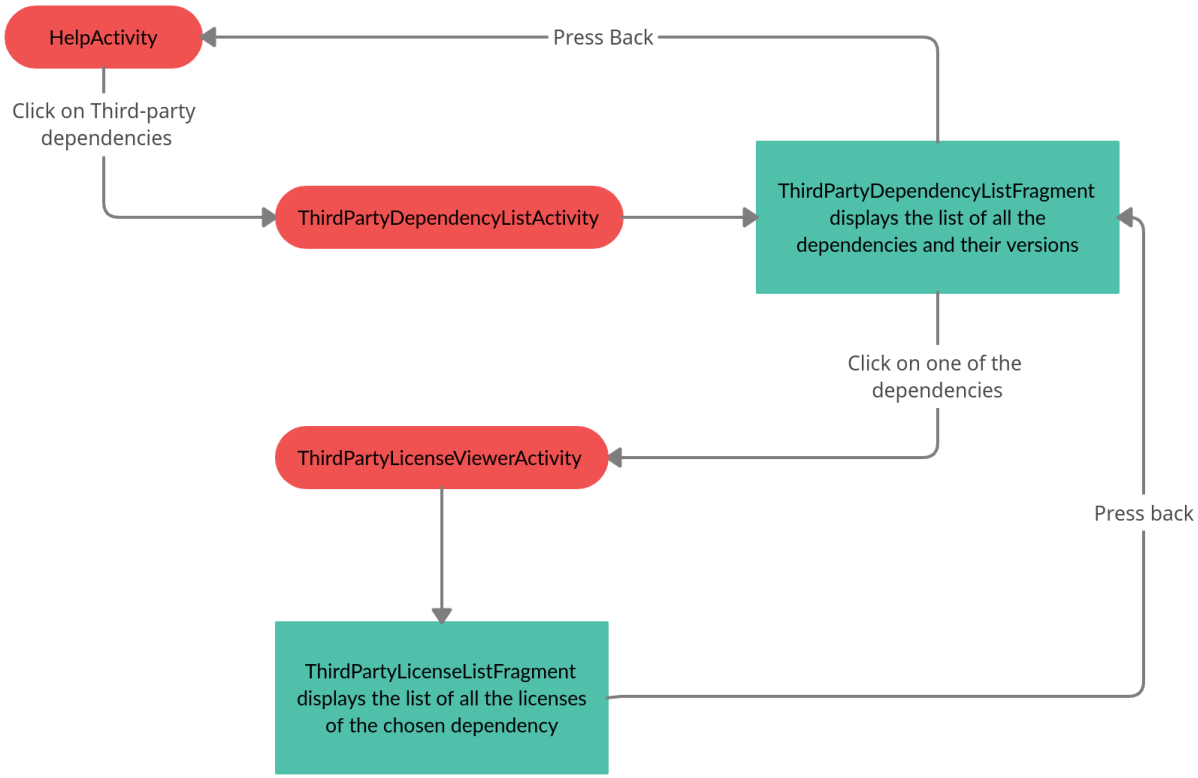
1. app/src/main/java/org/oppia/android/app/help/thirdparty
    - ThirdPartyDependencyListActivity/Presenter
        - This activity will be created when the user chooses the "Third-party Dependencies" option from the Help menu and it will also contain ThirdPartyDependencyListFragment.
    - ThirdPartyDependencyListFragment/Presenter
        - This fragment will contain a RecyclerView that will display the list of dependencies along with their versions and the the BindbleAdpater of this RecyclerView will use a SingleTypeBuilder as all the dependency items will have the same design.
    - ThirdPartyDependencyListViewModel
        - This ViewModel will provide the data list of the dependencies to the ThirdPartyDependencyListFragment.
    - RouteToLicenseViewerListener

        - This Interface will be implemented by the ThirdPartyDependencyListActivity to start the new activity to display the license texts.
    - ThirdPartyDependencyItemViewModel
        - This ViewModel will be responsible for the data (dependency name and version) handling of a single dependency item.
2. app/src/main/java/org/oppia/android/app/help/thirdparty/thirdpartylicense
    - ThirdPartyLicenseViewerActivity/Presenter
        - This activity will be created when the user clicks on one of the dependency items from the list displayed in the ThirdPartyLicenseViewerFragment.
    - ThirdPartyLicenseViewerFragment/Presenter
        - This fragment will contain a RecyclerView that will show the list of License texts for a single dependency.
    - ThirdPartyLicenseListViewModel
        - This ViewModel will provide the data (the list of licenses) to the RecyclerView of ThirdPartyLicensViewerFragment via BindableAdapter (SingleTypeBuilder).
    - ThirdPartyLicenseItemViewModel
        - This ViewModel will provide the data (the license name and text) to an individual item of the RecyclerView of the ThirdPartyLicenseViewerFragment.
3. app/src/main/res/values/thirdparty
    - third_party_dependecy_names.xml
        - This file will contain the dependencies' names as string resources and an array of all the string resources.
    - third_party_dependency_versions.xml
        - This file will contain the dependencies' versions as string resources and an array of all the string resources.
    - license_texts.xml
        - This file will contain the actual license texts of all the licenses as string resources.
    - dependency_license_text_array.xml
        - This file will contain one array corresponding to each dependency and the items of one array will only be those string resources (from the license_texts.xml) that correspond to the licenses of that dependency. It will also contain an array that will contain all the array resources of license texts.

4. app/src/main/res/layout/
    - third_party_dependency_list_activity.xml
        - This will be the layout for the ThirdPartyDependencyListActivity.
    - third_party_dependency_list_fragment.xml

        - This will be the layout for the ThirdPartyDependencyListFragment.
    - third_party_license_viewer_activity.xml
        - This will be the layout for the ThirdPartyLicenseViewerActivity.
    - third_party_license_viewer_fragment.xml
        - This will be the layout for the ThirdPartyLicenseViewerFragment.
    - third_party_dependency_item
        - This will be the layout for a single item of the RecyclerView of the ThirdPartyDependencyListFragment (i.e how the rows of the dependencies list will look like).
    - license_item
        - This will be the layout for a single item of the RecyclerView of the ThirdPartyLicenseViewerFragment (i.e how the rows of each license text will look like).

The flow of these UI components is summarized in the below diagram -



scripts package

The scripts would be runnable via bazel only as one of them would internally use bazel query commands to determine the list of dependencies and the other script depends on the results of the first. The two scripts would be added to the scripts package (scripts package is in the root of the repository) along the specified path -

1. scripts/

- generate_maven_dependencies.kts
    - This script will generate the list of all the Maven dependencies along with their versions and their license links on which the Oppia Android app depends. The generated list will then be stored in a JSON file that will be present in the third_party package (in the root of the repository).
- generate_license_texts.kts
    - This script would be used to extract the license texts for all the Maven dependencies present in the list that is generated by the first script.
- BUILD.bazel
    - This BUILD file will be created to configure the Kotlin scripts run via Bazel.

third_party package

A new file that will contain the list of dependencies that would be generated by the first script will get added to the third_party package of the repository -

1. third_party/
    - maven_dependencies.json
        - This file will contain the list of Maven dependencies for which the license text would be extracted and the structure of the file would be in JSON format.

root of the repository

A new file will that would contain all the necessary information about the Maven Dependencies will be created while adding support for pinning artifacts -

1. (root of the repository)/
    - maven_install.json
        - This will be an auto-generated file that will contain the information about the Maven dependencies used in the repository and will reside in the root of the repository.

.github directory

To implement the CI checks, we'll modify the static_checks.yml file in the workflows directory that will trigger these CI checks for every PR that is to be merged to the develop branch of the Oppia repository -

1. .github/workflows/
    - static_checks.yml
        - We will add a new job named "third_party_checks" in this file and this will verify that the list of dependencies is always up-to-date and the resource files generated by the second script can't be accidentally checked into Git.

Implementation Approach

The implementation of the project consists of various parts -

1. Introducing Kotlin script to compile the list of all the direct and indirect Maven dependencies along with their - versions and links (the links that would be utilized to extract the license text that) that are used in the project.
2. Introducing another Kotlin script to extract the license text for each of the dependencies that we get from step - 1 and also populate the strings XML resource files which will store the dependencies' - names, versions, and license texts.
3. Implementing UI to display the list of dependencies and their corresponding copyright license texts.
4. Implementing Github CI checks to verify that no Maven dependency is left for the license text extraction or vice versa and verify that the resource files generated via script-2 can never be checked in.

Introducing Kotlin Script to Compile Maven Dependencies

To compile the list of dependencies in a JSON file, we will first introduce the pinning_artifacts mechanism to our repository. rules_jvm_external supports pinning artifacts and their SHA-256 checksums into a `maven_install.json` file that can be checked into our repository.
Without artifact pinning, in a clean checkout of the project, rules_jvm_external executes the full artifact resolution and fetching steps (which can take a bit of time) and does not verify the integrity of the artifacts against their checksums. The downloaded artifacts also cannot be shared across Bazel workspaces.

By pinning artifact versions, we can get improved artifact resolution and build times, since using `maven_install.json` enables rules_jvm_external to integrate with Bazel's downloader that caches files on their SHA-256 checksums. It also improves resiliency and integrity by tracking the SHA-256 checksums and original artifact URLs in the JSON file.

In simpler words, after we introduce pinning artifact versions into our repository, a maven_install.json file will be created that will contain the information (like artifact name, dependency-jar URL, SHA-256 checksum, etc) about the Maven dependencies on which the Oppia-android app depends, and this file will also help us in providing versions and the URLs of the POM file of the Maven dependencies.

To get started with pinning artifacts, we will run the following command manually to generate the initial `maven_install.json` at the root of our Bazel workspace (since the `WORKSPACE` file is located in the root of the repository, the generated `maven_install.json` file will also be present in the root of the repository):

```
$ bazel run @maven//:pin
```

After this, we'll modify the maven_install() part of our `WORKSPACE` file -

Before

```
maven_install(
   artifacts = DAGGER_ARTIFACTS + get_maven_dependencies(),
   repositories = DAGGER_REPOSITORIES + [
       "https://bintray.com/bintray/jcenter",
       "https://jcenter.bintray.com/",
       "https://maven.fabric.io/public",
       "https://maven.google.com",
       "https://repo1.maven.org/maven2"
   ],
)
```

After

```
fail_if_repin_required = True

maven_install(
   artifacts = DAGGER_ARTIFACTS + get_maven_dependencies(),
   maven_install_json = "//:maven_install.json",
   repositories = DAGGER_REPOSITORIES + [
       "https://bintray.com/bintray/jcenter",
       "https://jcenter.bintray.com/",
       "https://maven.fabric.io/public",
       "https://maven.google.com",
       "https://repo1.maven.org/maven2"
   ],
)

load("@maven//:defs.bzl", "pinned_maven_install")

pinned_maven_install()
```

After this, the `maven_install.json` will become the new source of truth for maven dependencies, e.g, if a dependency is added/deleted/updated in the `versions.bzl` file, the changes will not be reflected in the repository. So, we'll need to update the `maven_install.json` file whenever we make some changes in the versions.bzl file. To update the `maven_install.json` file manually, we can run the following command -

```
$ bazel run @unpinned_maven//:pin
```

But we'll automate this step by running it via the first script only because whenever there will be some changes in `maven_install.json`, we'll always want to recompile the list of dependencies.
The generated file `maven_install.json` will have a similar structure to the below-mentioned snippet -


```
{
   "dependency_tree": {
       "__AUTOGENERATED_FILE_DO_NOT_MODIFY_THIS_FILE_MANUALLY": -473716691,
       "conflict_resolution": {
           "androidx.appcompat:appcompat:1.0.2": "androidx.appcompat:appcompat:1.2.0",
           "androidx.core:core:1.0.1": "androidx.core:core:1.3.0"
       },
       "dependencies": [
           {
               "coord": "android.arch.core:common:1.1.1",
               "dependencies": [
                   "com.android.support:support-annotations:28.0.0"
               ],
               "directDependencies": [
                   "com.android.support:support-annotations:28.0.0"
               ],
               "file":
"v1/https/maven.google.com/android/arch/core/common/1.1.1/common-1.1.1.jar",
               "mirror_urls": [
        "https://maven.google.com/android/arch/core/common/1.1.1/common-1.1.1.jar"
               ],
               "sha256": "3a616a32f433e9e23f556b38575c31b013613d3ae85206263b7625fe1f4c151a",
               "url":
"https://maven.google.com/android/arch/core/common/1.1.1/common-1.1.1.jar"
           },
           {
               "coord": "android.arch.core:core-testing:1.1.1",
               "dependencies": [
                   "junit:junit:4.12",
                   "org.hamcrest:hamcrest-core:1.3",
               ],
               "directDependencies": [
                   "android.arch.core:runtime:aar:1.1.1",
               ],
               "file":
"v1/https/maven.google.com/android/arch/core/core-testing/1.1.1/core-testing-1.1.1.aar",
               "mirror_urls": [
"https://maven.google.com/android/arch/core/core-testing/1.1.1/core-testing-1.1.1.aar"
               ],
               "sha256": "150a6f028af6c57ca6de66cb185a135cdcf737f597443acc4cf8aeeec064fff3",
               "url":
```

```
"https://maven.google.com/android/arch/core/core-testing/1.1.1/core-testing-1.1.1.aar"
          }
      ]
   }
}
```

After we successfully introduce the pinning_artifacts mechanism, we'll now create the first script that will use the bazel query language to find the list of all Maven dependencies (direct + transitive). We'll run a bazel query command via this script only and store its output in a string variable. The bazel query command will look something like this -

```
$ bazel query 'deps(deps(//:oppia) intersect //third_party/...) intersect
@maven//...'
```

The above query first takes the intersection* of the dependencies on which the Oppia app depends and the dependencies that are also part of third_party's `BUILD.bazel` (this gives us all the direct Maven dependencies on which Oppia Android app depends), then we find the dependencies on which the previous result depends and take an intersection* with the Maven dependencies (this step includes all the indirect/transitive dependencies). This way the adobe query will provide us with the complete list of Maven Dependencies on which Oppia-android depends.

***Note** - Intersection means the common subset of the two or more sets.

The output of this command will look like this (the actual list will be very large, this is just an example) -

```
@maven//:nl_dionsegijn_konfetti
@maven//:nl_dionsegijn_konfetti_1_2_5_extension
@maven//:javax_annotation_javax_annotation_api
@maven//:javax_annotation_javax_annotation_api_1_3_2_extension
@maven//:de_hdodenhof_circleimageview
@maven//:de_hdodenhof_circleimageview_3_0_1_extension
@maven//:com_squareup_retrofit2_converter_moshi
```

Let's assume that the output of this command is stored in a string named `dependencies_list`. All the dependencies in this string will be separated by a space (it may be possible that these are separated by a next line character i.e '\n' but this won't cause any change in the structure of code), and hence we will store the characters iterated until space is encountered in a separate string and store this set of characters as a single dependency name to be searched in the `maven_install.json`.

We'll create a set of dependencies listed in `maven_install.json` with the help of Moshi, and while creating the set, we'll also apply some transformations on the names of the dependencies so that they can be easily mapped to those that we get from the bazel query. The transformation will include - replacing every dot (' . ') and colon (' : ') with an underscore (' _ ') and omitting the version from the name.
For example, the dependency `"android.arch.core:core-testing": "1.1.1"` will be referenceable with - `android_arch_core_core-testing`.

Then we can search for each of the dependencies (that we get from the bazel query command) in this set. Once we get the dependency node in the set, we look for the "url" key in it. This url corresponds to the .jar file or .aar file of the dependency and we can generate the link of the POM file just by replacing the extension with .pom.
For example,
https://repo1.maven.org/maven2/com/almworks/sqlite4java/sqlite4java/0.282/sqlite4java-0.282.jar corresponds to the jar of the dependency and
https://repo1.maven.org/maven2/com/almworks/sqlite4java/sqlite4java/0.282/sqlite4java-0.282.pom corresponds to the POM file of the dependency.

The POM file of a Maven dependency is an XML file that contains information about the dependency and configuration details used by Maven to build that dependency. Most of the POM files have a `<licenses>` tag that contains the link(s) to its copyright license. An example is shown below -

```
<!-- This code snippet is a part of the POM file of the Glide dependency -->
<name>Glide</name>
<description>
  A fast and efficient image loading library for Android focused on smooth
  scrolling.
</description>
<url>https://github.com/bumptech/glide</url>
<licenses>
  <license>
    <name>Simplified BSD License</name>
    <url>http://www.opensource.org/licenses/bsd-license</url>
    <distribution>repo</distribution>
  </license>
  <license>
    <name>The Apache Software License, Version 2.0</name>
    <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
    <distribution>repo</distribution>
  </license>
</licenses>
```

We will extract the license links from the POM file URL of the dependency by reading its contents and picking up text from the `<url>` tags under the `<license>` tags and create an array of license links in the `maven_dependencies.json`.

Also, it isn't necessary that the POM file of each dependency contains a `<licenses>` tag. In this case, the script will fail (which further means that the CI check will also fail) and will need to build other feasible mechanisms for such cases or we'll have to replace such dependencies.

The generated list of dependencies that will be stored in `maven_dependencies.json` will have a similar structure to the below snippet -

```
{
 "third_party_dependencies": [
   {
     "id": 1,
     "name": "androidx.databinding:databinding-compiler-common",
     "version": "3.4.2",
     "license_links": [
       "http://www.apache.org/licenses/LICENSE-2.0.txt",
       "http://www.opensource.org/licenses/bsd-license"
     ],
     "custom_oppia_license_ids": [
       "bsd_license_0"
     ]
   },
   {
     "id": 2,
     "name": "com.crashlytics.sdk.android:answers:aar",
     "version": "1.4.6",
     "license_links": [
       "http://www.apache.org/licenses/LICENSE-2.0.txt"
     ],
     "custom_oppia_license_ids": []
   }
 ]
}
```

We already use a third-party library called Moshi (a third-party library that makes JSON parsing easier) to parse the JSON in our oppia-android repository, so storing the list of dependencies in a JSON structure will smoothen the parsing of this list when implementing the CI checks for this feature.

Structure of the data classes that can be potentially used to parse the dependencies list -

```
/** Data class for the list of Maven Dependencies. */
@JsonClass(generateAdapter = true)
data class MavenDependenciesList(
  @Json(name = "third_party_dependencies_list")
  val thirdPartyDependenciesList: List<MavenDependencies>
)
```

```
/** Data class that contains all info about a single Maven Dependency. */
@JsonClass(generateAdapter = true)
data class MavenDependencies(
  @Json(name = "id") val id: Int,
  @Json(name = "name") val name: String,
  @Json(name = "version") val version: String,
  @Json(name = "license_links") val licenseLinks: List<String>,
  @Json(name = "custom_oppia_license_ids")
  val customOppiaLicenseIds: List<String>
)
```

Introducing Kotlin Script to Extract License Texts

After we have generated the list of all the Maven Dependencies along with their - names, versions, and licenses' links, we will scrape the license texts for each dependency in the list by making use of the license links mentioned in the `maven_dependencies.json` and will also generate the string XML resource files that will contain the dependencies' names, versions, and the license texts. These resource files would be used to embed the license texts in the UI.

It is also possible that the URL of the license text does not contain plain text.
For example, BSD License mentioned in the above code snippet, the URL is mentioned below -
https://opensource.org/licenses/bsd-license

To handle these corner cases, we need to have a backup mechanism that would ensure that the license texts can always be extracted. We will maintain a separate repository on Github that will contain such license texts and we can scrape the text from there. We will follow the below steps to extract the license text from the new Oppia repository (assuming the name of the new repository to be "**oppia-android-licenses**") -

1. We will start with the base URL of the new Oppia repository - https://github.com/oppia/oppia-android-licenses
2. This URL can be modified to point to a file of the repository by appending the string - "/blob/{default_branch_name}/{path_of_the_file_to_be_viewed}"

To map the correct license file, we maintain a separate attribute - `custom_oppia_license_ids` that will give us the list of licenses for which text needs to be extracted from the `oppia-android-licenses` repository.
Assuming that the default branch name will be `develop` and the example license id to be `bsd_license_0` that lies in the root of the repository, the new URL becomes - https://github.com/oppia/oppia-android-licenses/blob/develop/bsd_license_0.txt
Note that we append the ".txt" extension to the id of the license.

3. But this URL wouldn't contain license text as plain text, to convert this URL to display the text as plain text, we would replace "github" with "raw.githubusercontent" in the URL. So, the final URL becomes - https://raw.githubusercontent.com/oppia/oppia-android-licenses/blob/develop/bsd-license.txt
4. Now we can simply scrape the license text from the above URL.

To scrape the text from a given URL, we can simply open the URL using openStream() and read the lines with the help of bufferReader as follows -

```
// All the scraped text would get stored as a single string in the variable
// named `licenseText`.
val licenseText = URL(url).openStream().bufferedReader().readLines()
```

After scraping the license text for a particular dependency, the script will write the dependency's name, version, and licenses' names in the corresponding resource files.
Dependency's name and version will be provided by the list that is generated by the first script, and the license name can be obtained from the `<name>` tag (it is present inside the `<license>` tag). The generated resource files will look similar to -

- third_party_dependency_names.xml

```
<!-- Dependencies' names. -->
<string name="third_party_dependency_name_0">Glide</string>
<string name="third_party_dependency_name_1">Konfetti</string>
<string name="third_party_dependency_name_2">Retrofit</string>

<!-- Array that stores dependencies' names. -->
<string-array name="third_party_dependency_names">
 <item>@string/third_party_dependency_name_0</item>
 <item>@string/third_party_dependency_name_1</item>
 <item>@string/third_party_dependency_name_2</item>
</string-array>
```

- third_party_dependency_versions.xml

```
<!-- Dependencies' versions. -->
<string name="third_party_dependency_version_0">4.3.1</string>
<string name="third_party_dependency_version_1">1.2.2</string>
<string name="third_party_dependency_version_2">1.3.4</string>

<!-- Array that stores dependencies' versions. -->
<string-array name="third_party_dependency_versions">
 <item>@string/third_party_dependency_version_0</item>
 <item>@string/third_party_dependency_version_1</item>
 <item>@string/third_party_dependency_version_2</item>
</string-array>
```

- license_texts.xml - We'll use CDATA for the license texts to maintain the format of the license text and to ignore any angular brackets (' < ' or ' > ') while XML parsing.

```
<!-- License texts. -->
<!-- These are just samples and not complete license texts. -->
<string name="bsd_license">
<![CDATA[
Copyright 2021,
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
]]>
</string>
<string name="apache_license_2">
<![CDATA[
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.
```

```
      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.
]]>
</string>
```

- dependency_license_text_arrays.xml

```
<!-- Arrays that store licenses' names. -->
<string-array name="dependency_licenses_0">
 <item>@string/bsd_license</item>
 <item>@string/apache_license_2</item>
</string-array>

<string-array name="dependency_licenses_1">
 <item>@string/apache_license_2</item>
</string-array>

<!-- Array that stores arrays of licenses for each dependency. -->
<array name="dependency_licenses_texts">
 <item>@array/dependency_licenses_0</item>
 <item>@array/dependency_licenses_1</item>
</array>
```

We can write files in Kotlin by using bufferedWriter(), an example is shown below -

```
val filePath = "/some/path/fileName"
val text = "This is some text to be appended\n"
FileOutputStream(filePath, true).bufferedWriter().use { writer ->
     writer.append(text)
}
```

We will iterate over the list of dependencies generated by the first script and generate the license text for each dependency and also write the resource XML files accordingly. In the above example, we write the file in append mode by passing the parameter "append" of FileOutputStream as `true`. In our case, we will always append the text to generate these resource files. But there is one exception when the script is run again, we want to erase the old content and write it from scratch. To do this, we can create a counter that will count the number

of dependencies processed till now (for how many dependencies we have written the files) and check if the counter is 0 or not. If the counter is 0, that means the script is going to process the first dependency and the script is being run again (or the very first time) so we will not write these files in append mode when the counter is 0. If the counter is non-zero that means we are in between of the process and we always want to write the files in append mode in this case.

Implementing UI

The UI implementation is broken down to specify details at the individual file level. The below-mentioned UI components will be modified as follows -

HelpItems

The enum HelpItems will now contain a new constant ordinal to support displaying Third-party libraries as an option in the HelpFragment.

Before	After

`enum class HelpItems {`
` FAQ;`
`}`
	`enum class HelpItems {`
	` FAQ,`
	` THIRD_PARTY;`
	`}`

HelpListViewModel

The HelpListViewModel's getRecyclerViewItemList() method will be modified to contain a when clause instead of an if statement so that we can also add the item "Third-party Dependencies" to the list of helpFragmentRecyclerView.

```
private fun getRecyclerViewItemList(): ArrayList<HelpItemViewModel> {
 for (item in HelpItems.values()) {
   val category: String
   val helpItemViewModel: HelpItemViewModel
   when (item) {
     HelpItems.FAQ -> {
       category =
activity.getString(R.string.frequently_asked_questions_FAQ)
       helpItemViewModel = HelpItemViewModel(activity, category)
     }
     HelpItems.THIRD_PARTY -> {
       category = activity.getString(R.string.third_party_libraries)
       helpItemViewModel = HelpItemViewModel(activity, category)
     }
   }
   arrayList.add(helpItemViewModel)
```

```
 }
 return arrayList
}
```

HelpItemViewModel

The HelpItemViewModel's onClick() method will be modified to handle the cases when the user clicks on the "Third-party libraries" item. We'll replace the if statement with a when clause here too.

```
fun onClick(title: String) {
 when (title) {
   activity.getString(R.string.frequently_asked_questions_FAQ) -> {
     val routeToFAQListener = activity as RouteToFAQListListener
     routeToFAQListener.onRouteToFAQList()
   }
   activity.getString(R.string.third_party_libraries) -> {
     val routeToThirdPartyListener = activity as RouteToThirdPartyListener
     routeToThirdPartyListener.onRouteToThirdPartyList()
   }
 }
}
```

The following new UI components will be added to display the list of dependencies and their License texts -

ThirdPartyDependencyListActivity and RouteToLicenseViewerListener

- The RouteToLicenseViewerListener will be an interface that will be implemented by ThirdPartyDependencyListActivity.
- The interface will contain a method that will be overridden by the ThirdPartyDependencyListActivity to create the Intent for the ThirdPartyLicenseListActivity so that the ThirdPartyLicenseListActivity can be created when the RouteToLicenseViewerListener calls its method.

ThirdPartyDependencyActivityPresenter

- This presenter will handle all the methods of the ThirdPartyDependencyListActivity. We will inject ThirdPartyDependencyListActivityPresenter in the ThirdPartyDependencyListActivity via field dependency injection (Dagger dependency injection) and the original methods of the ThirdPartyDependencyListActivity will just call the ThirdPartyDependencyListActivityPresenter's methods.

- We also launch the ThirdPartyDependencyListFragment as soon as the ThirdPartyDependencyListActivity is created to display the list of dependencies to the users.
- An example is given below -
  onCreate() method overridden in ThirdPartyDependencyListActivity -

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  activityComponent.inject(this)
 // onCreate() method contains only necessary lines of
 // code and rest of the work is done by
 //`thirdPartyDependencyListActivityPresenter.handleOnCreate()`
  thirdPartyDependencyListActivityPresenter.handleOnCreate()
}
```

handleOnCreate() method containing all other code -

```
fun handleOnCreate() {
 val binding =

DataBindingUtil.setContentView<ThirdPartyDependencyListActivityBinding>(act
ivity, R.layout.third_party_dependency_list_activity)
 binding.apply {
   lifecycleOwner = activity
 }
 // As soons as the ThirdPartyDependencyListActivity is created, we launch
 // the ThirdPartyDependencyListFragment to display the list of
 //dependencies.
 if (getThirdPartyDependencyListFragment() == null) {
   activity.supportFragmentManager.beginTransaction().add(
     R.id.third_party_dependency_list_fragment_placeholder,
     ThirdPartyDependencyListFragment()
   ).commitNow()
 }
}
```

ThirdPartyDependencyListFragment, ThirdPartyDependencyListViewModel and ThirdPartyDependencyItemViewModel

- This fragment will be displayed as soon as the ThirdPartyDependencyListActivity will be created and it will display the list of dependencies and their versions.

- This fragment will contain a RecyclerView to show the list of dependencies and the data of this list will be provided by the ThirdPartyDependencyListViewModel.
- ThirdPartyDependencyItemViewModel will contain the data that will be bound to a single item of the RecyclerView of the ThirdpartyDependencyListFragment.
- The implementation of ThirdPartyDependencyListViewModel will look similar to the below-provided code snippet -

```
class ThirdPartyDependencyListViewModel @Inject constructor(
 val activity: AppCompatActivity
) : ObservableViewModel() {
 private val arrayList = ArrayList<ThirdPartyDependencyItemViewModel>()

 val thirdPartyDependencyItemList: List<ThirdPartyDependencyItemViewModel>
by lazy {
   getRecyclerViewItemList()
 }

 private fun getRecyclerViewItemList():
ArrayList<ThirdPartyDependencyItemViewModel> {
   val third_party_dependecy_names: Array<String> =
  activity.resources.getStringArray(R.array.third_party_dependency_names)
   val third_party_dependencies_versions: Array<String> =
     activity.resources.getStringArray(
     R.array.third_party_dependency_versions
)
   third_party_dependency_names.forEachIndexed { index, name ->
     val thirdPartyDependencyItemViewModel =
       ThirdPartyDependencyItemViewModel(activity, name,
third_party_dependency_versions[index])
     arrayList.add(thirdPartyDependencyItemViewModel)
   }
   return arrayList
 }
}
```

- In the above code snippet, we iterate over the array that stores the dependencies' names and add each dependency's name and version to the arrayList (the version is retrieved from the third_party_dependency_versions array as we also maintain the index of the array while iterating on the third_party_dependency_names array).

ThirdPartyDependencyListFragmentPresenter

- All the methods of the ThirdPartyDependencyListFragment will be handled by ThirdPartyDependencyListFragmentPresenter e.g, onCreateView(), onAttach() etc.
- The adapter of the RecyclerView of ThirdPartyDependencyListFragment will also be created here -

```
// This function returns an instance of the BindableAdapter that will be
// attached to the RecyclerView of the ThirdPartyDependencyListFragment.
private fun createRecyclerViewAdapter():
BindableAdapter<ThirdpartyDependencyItemViewModel> {
    return BindableAdapter.SingleTypeBuilder
      .newBuilder<ThirdpartyDependencyItemViewModel>()
      .registerViewDataBinderWithSameModelType(
         inflateDataBinding = ThirdPartyDependencyItemBinding::inflate,
         setViewModel = ThirdPartyDependencyItemBinding::setViewModel
      )
      .build()
}
```

ThirdPartyLicenseViewerActivity and ThirdPartyLicenseViewerActivityPresenter

- ThirdPartyLicenseViewerActivity will be created when the user clicks on one of the items of the RecyclerView of the ThirdPartyDependencyListFragment and it will also contain the ThirdPartyLicensViewerFragment to show the list of License texts of the clicked dependency item.
- ThirdPartyLicenseViewerActivityPresnter will handle all the methods of the ThirdPartyLicenseViewerActivity.
- ThirdPartyLicenseViewerActivity will also have a companion object that will contain a method to create the intent for it and will also allow us to retrieve the index of the dependency item on which the user clicked -

```
companion object {
 const val THIRD_PARTY_LICENSE_VIEWER_ACTIVITY_ITEM_INDEX =
"ThirdPartyLicenseViewerActivity.itemIndex"

 fun createThirdPartyLicesneViewerActivityIntent(context: Context, index:
Int): Intent {
   val intent = Intent(context,
ThirdPartyLicenseViewerActivity::class.java)
   intent.putExtra(THIRD_PARTY_LICENSE_VIEWER_ACTIVITY_ITEM_INDEX, index)
   return intent
```

```
 }
}
```

ThirdPartyLicenseViewerFragment and ThirdPartyLicenseViewerFragmentPresenter

- ThirdPartyLicenseViewerFragment will display the list of license texts for a dependency and thus it will again contain a recyclerview that will show the license text items.
- ThirdPartyLicenseViewerFragmentPresenter will handle all its methods and will also create the BindableAdapter for its RecyclerView.
- The implementation will be very similar to the RecyclerView of the ThirdPartyLicenseViewerFragment -

```
// This function returns an instance of the BindableAdapter that will be
// attached to the RecyclerView of the ThirdPartylicenseViewerFragment.
private fun createRecyclerViewAdapter():
BindableAdapter<ThirdpartyLicenseItemViewModel> {
    return BindableAdapter.SingleTypeBuilder
      .newBuilder<ThirdpartyLicenseItemViewModel>()
      .registerViewDataBinderWithSameModelType(
         inflateDataBinding = ThirdPartyLicenseItemBinding::inflate,
         setViewModel = ThirdPartyLicenseItemBinding::setViewModel
      )
      .build()
}
```

ThirdPartyLicenseListViewModel and ThirdPartyLicenseItemViewModel

- ThirdPartyLicenseListViewModel will provide the list of licenses to the ThirdPartyLicenseViewerFragment.
- ThirdPartyLicenseItemViewModel will provide the data of the single license item to the ThirdPartyLicenseViewerFragment.
- The implementation of ThirdPartyLicenseListViewModel will look like this -

```
class ThirdPartyLicenseListViewModel @Inject constructor(
 val activity: AppCompatActivity,
 val licenseIndex: Int
) : ObservableViewModel() {
 private val arrayList = ArrayList<ThirdpartyLicenseItemViewModel>()

 val thirdPartyLicenseItemList: List<ThirdPartyLicenseItemViewModel> by
lazy {
   getRecyclerViewItemList()
```

```
 }

private fun getRecyclerViewItemList():
ArrayList<ThirdPartyLicenseItemViewModel> {
   val dependencies_license_texts: Array =
     activity.resources.getArray(R.array.dependencies_license_texts)
   val licenses: Array<String> = dependencies_license_texts[licenseIndex]
   licenses.forEachIndexed { index, license_text ->
     val thirdPartyLicenseItemViewModel =
       ThirdPartyLicenseItemViewModel(activity, license_text)
     arrayList.add(thirdPartyLicenseItemViewModel)
   }
   return arrayList
 }
}
```

XML Layout files

- third_party_dependency_list_activity.xml
- third_party_dependency_list_fragment.xml
- third_party_license_viewer_activity.xml
- third_party_license_viewer_fragment.xml
- third_party_item
- license_item

These files will be implemented according to the mocks.

Implementing Github CI checks

The list of the Maven dependencies generated by the first script should always be up-to-date (it should neither include any extra Maven dependencies that are no longer present in the repository nor miss any Maven dependencies that are present in the repository). We need to implement a Github CI check that would fail if the list is missing dependencies (or containing some extra dependencies) and it would pass if the list is up-to-date.
Also, we want another CI check to verify that we never include the actual license texts to the Oppia-android repository (the actual license texts would be generated only by the maintainers while preparing for the release). We need to ensure that the results of script-2 (i.e the resource files generated by the second script) never get checked into VCS even if some contributor runs the second script locally by mistake.

Check to ensure that the dependencies list is always up-to-date

The list of dependencies generated by the first script would be incorrect if a contributor in some PR makes changes in the version of some Maven dependencies or add/delete some Maven

dependencies and then commit and push the changes to their fork without following the proper steps of updating dependencies.
The list of dependencies would be incorrect in 2 cases -

- **When the maven_install.json file is not up-to-date:**
  A contributor must run the command `$ bazel run @unpinned_maven//:pin` to reflect the changes of the versions.bzl file in the whole project as maven_install.json will be the new source of truth for all the Maven dependencies used in the repository. For example, if we add a dependency and do not run the above command, then we shall not be able to actually use the dependency's functionalities in our codebase.
  So, to keep the list of dependencies up-to-date, we need to make sure that `maven_install.json` is always up-to-date. If we forget to update the maven_install.json, normally, rules_jvm_external will print a warning to the console and continue the build when this happens, but we will prevent this by setting the fail_if_repin_required attribute to True, when this attribute is set and the list isn't up-to-date, this will be treated as a build error, causing the build to fail. When this attribute is set, it is possible to update the `maven_install.json` file using:
  `$ REPIN=1 bazel run @unpinned_maven//:pin`

  So, we set the fail_if_repin_required attribute in the `WORKSPACE` file to True, which will cause the Build to fail and we already have a CI check that ensures that we are always able to build the app using bazel, so we will always be able to catch this.
  Also, to avoid the contributors from running the command manually, we'll include this command in the first script itself (as running the first script will be a necessary step after adding/deleting/updating Maven dependencies).

- **When the first script is not run at all after updating versions.bzl:**
  The contributor must run the first Kotlin script to update the list of Maven dependencies. To check this, we will make two lists of Maven dependencies - one from the `maven_dependencies.json` file of the contributor's branch, and the other list would be generated by running the first script in the CI (since the first script relies on maven_install.json for the generation of dependencies list, it would always be up-to-date as we'll make sure that the `maven_install.json` is up-to-date by running the command to update it through the script only). Now we will subtract both the lists and find the difference between the two. If the difference is non-empty, we will fail the CI check and will also call out those dependencies which appear to be in the difference of the two lists. Implementation of this step will look something like this (the below snippet is not exact and it just makes the idea clear, the actual implementation will be slightly complicated as the lists will be generated by two different methods) -

```
// Let's consider the second list is always correct in this example.
val list1 = arrayListOf(1, 2, 3, 4, 5)
val list2 = arrayListOf(3, 4, 5, 6)
val difference1 = list1 - list2
```

```
val difference2 = list2 - list1
val checkPassed: Boolean = difference1.isEmpty() && difference2.isEmpty()
if (checkPassed) {
 println("The list of Maven dependencies is up-to-date.")
} else {
 println("The list of Maven dependencies needs to be updated -")
 if (difference1.isNotEmpty()) {
   println("The following dependencies were found redundant -
$difference1")
 }
 if (difference2.isNotEmpty()) {
   println("The following dependencies were found missing - $difference2")
 }
}
```

We can utilize the first script to implement this check by passing a few arguments to the script (these arguments would act as flags determining where the script is being run) to the script that would modify the script behavior. If the script is being run on a local machine, the script would generate a JSON file listing all dependencies' names, versions, and license links and if it is being run on CI, we can just save the list in a variable and create another list by iterating over the `maven_dependencies.json` file of the contributor's branch. And then we can check the difference between the two lists and make the check pass/fail based on that.

Check to ensure that the generated resource files never get checked-in

The resource files generated by the second script should never be edited by any contributor in their PR and even if they do so by mistake, we need to detect this and tell the contributor that these files should not be touched and they need to fix this.

To prevent this, we'll add some comments manually when the resource files are generated for the very first time. And we know that whenever the script will run it will erase the old content of the resource files and then write the whole file again from scratch. So, the comments will also get erased when the second script will be run. To ensure that the second script is never run we can simply check in CI if the comments are present in the resource files (generated by the second script) or not.

To implement this we can create a new script similar to the lint_check scripts and we can run that script in the CI, if the script's exit code is 0 then the check will pass and if the exit code is non-zero, the check will fail. We can also simply add the commands directly to the `static_checks.yml` but having a script will look cleaner.

A new shell script `verify_second_script_not_run.sh` will be created which will check whether comments are present for all the resource files generated by the second script or not, and we will make it run as a static check.

```
# static_checks.yml
third_party_checks:
 name: Verify Third-party maven_dependecncies.json is up-to-date and
license text is never checked-in
 runs-on: ubuntu-18.04
 Steps:
   - name: Check maven_dependecies.json is up-to-date
   - run: # Implement the first CI check here.

   - name: License Text check
   - run: |
       bash
/home/runner/work/oppia-android/oppia-android/scripts/verify_second_script_
not_run.sh $HOME
```

Third-party Libraries

Currently, the project's implementation doesn't need any third-party libraries.

Testing Approach

We will test the flow (from beginning to the end) of the ThirdPartyDependencyListActivity, ThirdPartyDependencyListFragment, ThirdPartyLicenseViewerActivity, and ThirdPartyLicenseViewerFragment by writing tests in Espresso and Robolectric. We will also add tests specifically to test RecyclerViews that will be introduced in the project.
We will also add some tests for both the scripts to ensure that the baseline expectations are always met and don't break in the future.
Except for these automated tests which shall be included in the codebase, we'll also do some manual testing to demonstrate that the CI checks work as expected or not. A few Test-only PRs will be explicitly created to verify this.

Milestones

Milestone 1

Key Objective: To introduce two scripts that generate the list of Maven dependencies and extract the license texts for all the Maven Dependencies and also generate the resource XML files that will be used to hook up the names, versions, and licenses of the dependencies to the actual UI of the app.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	Introduce Pinning Artifacts Mechanism	NA	Jun 09, 2021	Jun 13, 2021
1.2	Write documentation to include all the steps to be taken after adding/deleting/updating any Maven dependency in versions.bzl	1.1	Jun 10, 2021	Jun 13, 2021
1.3	Create a script to generate the list of Maven dependencies, versions, and license links	1.1	Jun 19, 2021	Jun 24, 2021
1.4	Modify documentation to include all the steps to be taken after adding/deleting/updating any Maven dependency in versions.bzl	1.3	Jun 24, 2021	Jun 26, 2021
1.5	Create another script to extract the license texts for all the Maven dependencies obtained from the script-1 and generate resource XML files	1.2	Jul 02, 2021	Jul 07, 2021

Milestone 2

Key Objective: To introduce the new components in the UI of the app that display the dependencies' names, versions, and license texts and Github CI checks verify that the list of dependencies is always up-to-date, extraction of licenses is always possible, and add additional tests to ensure that the generated list of dependencies can't get checked into git.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Hook-up UI to display the list of dependencies in the app	1.4	Jul 20, 2021	Jul 24, 2021
2.2	Hook-up complete UI to display the list of license texts when a user clicks on one of the dependencies from the dependency list	2.1	Jul 28, 2021	Aug 02, 2021
2.3	Verify the new piece of UI does not have any accessibility issues (if any)	2.1	July 29, 2021	Aug 03, 2021

2.4	Implement Github CI check to verify the dependency list is always up-to-date	1.1	Aug 01, 2021	Aug 05, 2021
2.5	[TEST-ONLY] Verify the CI check (No. - 2.4) is catching the different cases of properly	2.4	Aug 06, 2021	NA
2.6	Implement CI check to ensure that the list of dependencies can't be accidentally checked into Git.	1.3	Aug 06, 2021	Aug 10 , 2021
2.7	[TEST-ONLY] Verify the CI check (No. - 2.6) is catching the different cases of failure properly	2.6	Aug 06, 2021	NA

Additional Project-Specific Considerations

Privacy

This feature does not collect any user data as we only display the list of dependencies on which the Oppia-Android app depends and their Copyright license texts. So, this project does not deal with any privacy issues of the Oppia users.

Security

This feature does not provide any new opportunities for users to gain unauthorized access to user data as the list of dependencies and license texts will reside internally within the app and we will not face any security issues.

Accessibility (if user-facing)

The UI that would be introduced through this project would be built in such a way that accessibility checks pass for all the screens. The new UI will not include any complex interactions for users and will not probably impose any accessibility challenges, but as the license texts would be quite lengthy, we may face some challenges in passing the accessibility checks for the ThirdpartyLicenseViewerActivity, and thus we'll need to figure out if any separate work is needed to overcome this accessibility challenge.

Documentation Changes

After the project is completed, we will need to add a new page in our wiki, that will contain the steps that should be taken after a Maven dependency has been updated/added/deleted in some PR.

Ethics

Displaying the Copyright Licenses for the other software used to build our software is one of the best practices that should be followed while developing software.

Future Work

This project is limited to introducing support for displaying the copyright licenses for only Maven dependencies, but we should also extend this feature to display the copyright license texts for the Non-maven dependencies too.

Additional Information

Adding Proper Documentation

To avoid developers running into problems because of repining Maven dependencies, we'll add all the steps (to be done after updating any Maven dependency) to our wiki.

The steps that the developers will have to follow after they update any Maven dependency in the codebase are -

1. Update the dependency's artifact or version (or both) in the `BUILD` or `versions.bzl` files.
2. Run the first Kotlin script on their local fork. This script will first internally run the command `$ bazel run @unpinned_maven//:pin` so that the changes in the dependencies' artifacts or versions get reflected in the `maven_install.json` file and then recompile the list of dependencies.
3. Commit and push the changes and create/update the PR. If everything is done correctly, the 2 new Github CI checks (that will be added during the implementation of this project) will pass and if they fail, take a look at the reason for failure and repeat the above steps in the correct order again.

We can also add this build failure as a common Bazel Build error to the Frequent Errors and Solutions Page of our Wiki and we can just add the link to the above steps as the solution for this Build error.

Failing CI checks eagerly

The first CI check will ensure that the maven_install.json and the list of dependencies generated by the first script are always up-to-date or not. And thus, we'll fail this check-in 3 scenarios -

1. When the maven_install.json is not up-to-date.
2. When the maven_install.json is up-to-date but the list of dependencies is not up-to-date.

(1) can happen when a developer has neither run the maven-repining command nor the first

script (note that running the first script would repin the maven dependencies itself as the command to repin the dependencies will be included in the first script), so if this is the case, the Build will fail which means that the Build with Bazel CI check will also fail. So, we can just make this check a prerequisite for the new check, and we can just skip the new check (or even fail it) because the Build check will always have a greater priority.

(2) can happen when a developer has only run the maven-repinning command and they haven't run the first script, so definitely maven_install.json will get updated and the list of dependencies will still remain outdated. To handle this case, we will generate two lists in CI - first by iterating over the list of Maven dependencies of the contributor's branch and another by running the script in CI to generate a new list of dependencies (this list of dependencies will be up-to-date since it will make use of the maven_install.json which will be updated in this case) and we will find the difference of the two lists. If the difference will be non-empty then it will imply that the list of dependencies of the contributor's branch is not up-to-date and the script hasn't been run by the contributor and hence we'll fail the CI check.

The second CI check will ensure that the second script (that will generate resource XML files) is not run by the general developers. We will add some comments in all the resource XML files that would be generated by the second script, e.g -

```
<!-- This file is automatically generated by the script -
generate_licenses.kts.

Do NOT modify, delete, or commit to source control! -->
```

Whenever the second script will be run, it will first erase the whole content of the resource XML files and then write these files from scratch. Thus if some developer runs the second script, then these comments will get erased from the resource files. So, in the second CI check, we'll simply verify if these comments are present in those resource XML files or not, if they will not be present then we will fail the CI check and prompt the user to fix the changes.