GSoC 2021 Proposal

End-to-End Testing Support

About You

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

Oppia provides free education for kids through the internet with a great way of teaching (i.e Explorations) which gives students a great learning experience. There is a great team working for oppia and it was fun being a part of this team and contributing to the codebase. Contributing to oppia is going to impact many learning students. This makes me happy.
Oppia has a great codebase and architecture, besides all these oppia is a great community that helps the contributors get onboarded and help one another. It was a great experience helping out new contributors and code reviews.

I love writing Instrumentation tests in android, though when I started contributing to oppia I had no idea about writing tests once I started I loved writing tests in both Robolectric and Espresso. Now I'm excited to work on UiAutomator.

Prior experience

I started android development a year ago. I've worked with the following technologies Dagger-Hilt, Coroutines, Retrofit, Jetpack components, Espresso, etc. I've also published an app which is a replacement of a website shreify to experience how publishing an app works.

I've worked as an Android Developer Intern at *Jan Elaaj*, an online consultation company for 2 months. Their app was in Java in which I had some trouble with it as I was mainly focused on Kotlin for android development. Here I've worked on many features and fixed bugs also got to publish many releases.

Then I wanted to step into open-source for more experience as an Android Developer, I've contributed to **Tachiyomi** and successfully got 2 PR merged, though these were minor contributions I still feel proud of it as this app was impacting many users.

I have 91 Reputation on StackOverflow by answering a few android related Discussions or Questions and helping out other android developers.

When I started contributing to Oppia, I really liked the way Onboardig mentors are assigned and how other contributors help out each other which kept me motivated and I was able to contribute to oppia daily with issues, pull requests and code reviews, and frequently I got to learn many new topics like Protocol buffers, Bazel, Instrumentation Tests, Local Test, Dagger and most importantly the MVP + MVVM architecture. I've also started Contributing out of curiosity to check how the backend works on the web side. I've joined the CLAM team (Core Learning and Mentoring) and part of the OnBoarding team in Oppia android and the Stability and Data Handling team in Oppia web.

Merged PRs in Oppia-android:

- #2344 - Migrated ProfileInputView to TextInputLayout for AddProfileActivity with Espresso and Robolectric tests
- #2392 - Migrating to BindableAdapter for LanguageSelectionAdapter
- #2648 - Created a BUILD.bazel file for app/viewmodel
- #2233 - Adding bullet indentation for Bullet points
- #2289 - Seperate Robolectric Check provider

Merged PRs in Oppia-web:

- #11869 - Removing email access in admin-page
- #11999 - Fix E2E tests in core/tests/protractor/accessibility.js

Contact info and timezone(s)

Name - Syed Farees Hussain
Country - India
College - Jamia Millia Islamia, Delhi
Email - fareezzhussain@gmail.com
Github - https://github.com/FareesHussain
Discord - FareesHussain#8100
LinkedIn - https://www.linkedin.com/in/farees-hussain-286a08193
Timezone - Indian Standard Time (IST) / +5:30 GMT
Preferred method of communication - Hangouts, e-mail, and Gitter

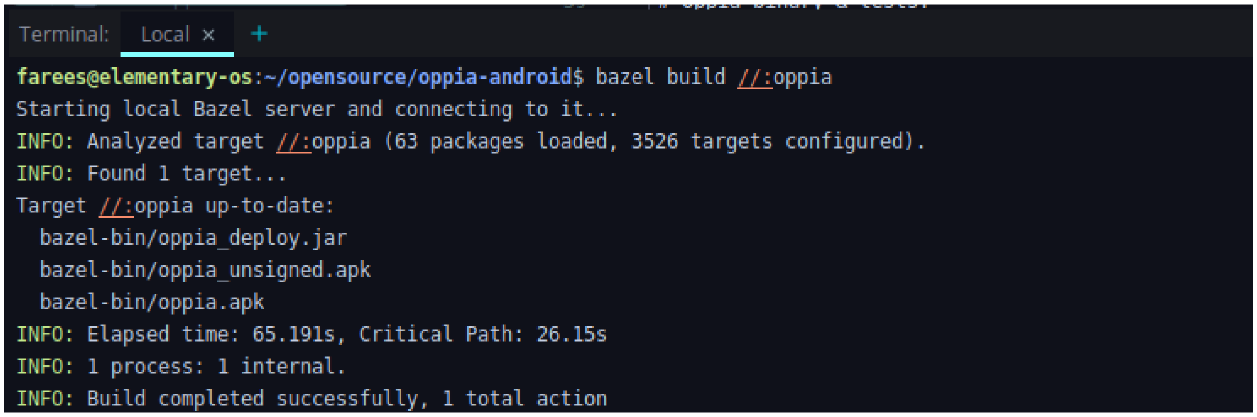
Time commitment

My time commitment to this project will be as follows -

- 7 hours a day (Monday - Saturday)
- Time dedication on Sunday will be subject to work requirements.
- Total: 40-50 hours a week.

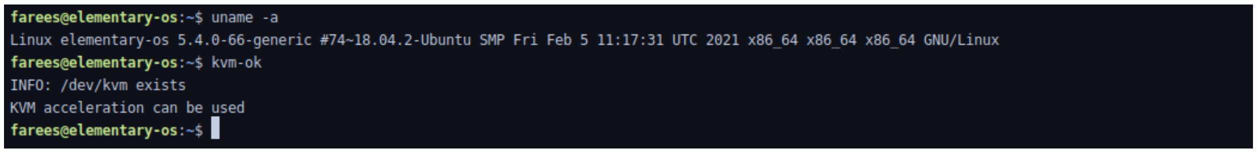
Essential Prerequisites

- I have submitted few pull requests that include Espresso tests #2195, #2264, #2344, #2345, #2349, #2350, #2510
- I am able to build Oppia-android using Bazel



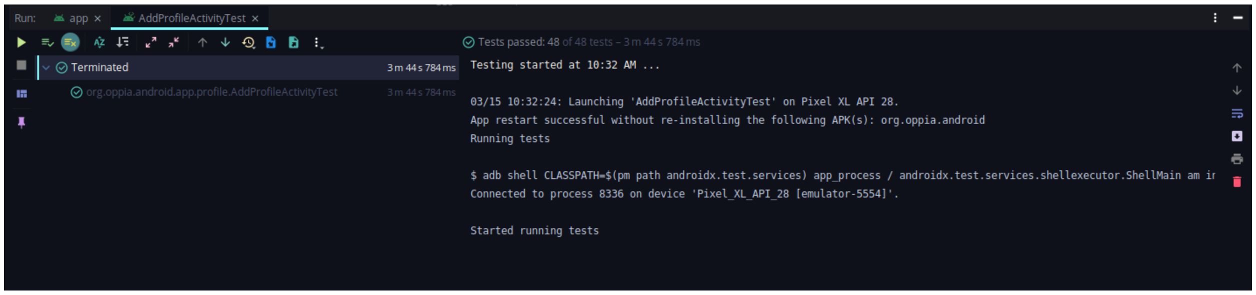
```
farees@elementary-os:~/opensource/oppia-android$ bazel build //:oppia
Starting local Bazel server and connecting to it...
INFO: Analyzed target //:oppia (63 packages loaded, 3526 targets configured).
INFO: Found 1 target...
Target //:oppia up-to-date:
  bazel-bin/oppia_deploy.jar
  bazel-bin/oppia_unsigned.apk
  bazel-bin/oppia.apk
INFO: Elapsed time: 65.191s, Critical Path: 26.15s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
```

- I use Linux and it supports KVM



```
farees@elementary-os:~$ uname -a
Linux elementary-os 5.4.0-66-generic #74~18.04.2-Ubuntu SMP Fri Feb 5 11:17:31 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
farees@elementary-os:~$ kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
farees@elementary-os:~$ 
```

- I am able to run all the Espresso tests of a test suite



Run:
app
AddProfileActivityTest
Terminated 3 m 44 s 784 ms
org.oppia.android.app.profile.AddProfileActivityTest 3 m 44 s 784 ms
Tests passed: 48 of 48 tests – 3 m 44 s 784 ms

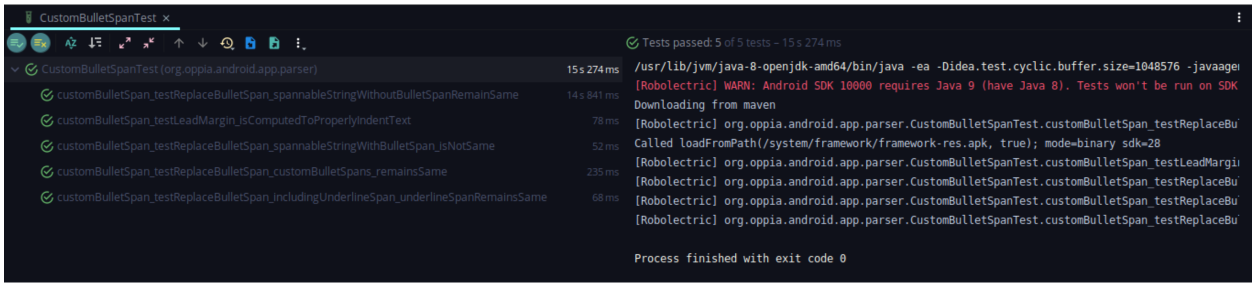
```
Testing started at 10:32 AM ...

03/15 10:32:24: Launching 'AddProfileActivityTest' on Pixel XL API 28.
App restart successful without re-installing the following APK(s): org.oppia.android
Running tests

$ adb shell CLASSPATH=$(pm path androidx.test.services) app_process / androidx.test.services.shellexecutor.ShellMain am ir
Connected to process 8336 on device 'Pixel_XL_API_28 [emulator-5554]'.

Started running tests
```

- I am able to run all Unit tests of a test suite

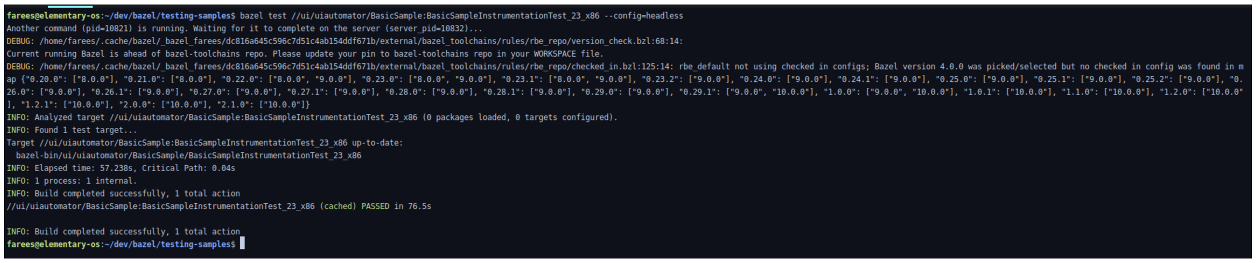


```
CustomBulletSpanTest (org.oppia.android.app.parser)                                                   15 s 274 ms
  customBulletSpan_testReplaceBulletSpan_spannableStringWithoutBulletSpanRemainSame                     14 s 841 ms
  customBulletSpan_testLeadMargin_isComputedToProperlyIndentText                                            78 ms
  customBulletSpan_testReplaceBulletSpan_spannableStringWithBulletSpan_isNotSame                            52 ms
  customBulletSpan_testReplaceBulletSpan_customBulletSpans_remainsSame                                     235 ms
  customBulletSpan_testReplaceBulletSpan_includingUnderlineSpan_underlineSpanRemainsSame                    68 ms
```

```
/usr/lib/jvm/java-8-openjdk-amd64/bin/java -ea -Didea.test.cyclic.buffer.size=1048576 -javaager
[Robolectric] WARN: Android SDK 10000 requires Java 9 (have Java 8). Tests won't be run on SDK
Downloading from maven
[Robolectric] org.oppia.android.app.parser.CustomBulletSpanTest.customBulletSpan_testReplaceBu
Called loadFromPath(/system/framework/framework-res.apk, true); mode=binary sdk=28
[Robolectric] org.oppia.android.app.parser.CustomBulletSpanTest.customBulletSpan_testLeadMargi
[Robolectric] org.oppia.android.app.parser.CustomBulletSpanTest.customBulletSpan_testReplaceBu
[Robolectric] org.oppia.android.app.parser.CustomBulletSpanTest.customBulletSpan_testReplaceBu
[Robolectric] org.oppia.android.app.parser.CustomBulletSpanTest.customBulletSpan_testReplaceBu

Process finished with exit code 0
```

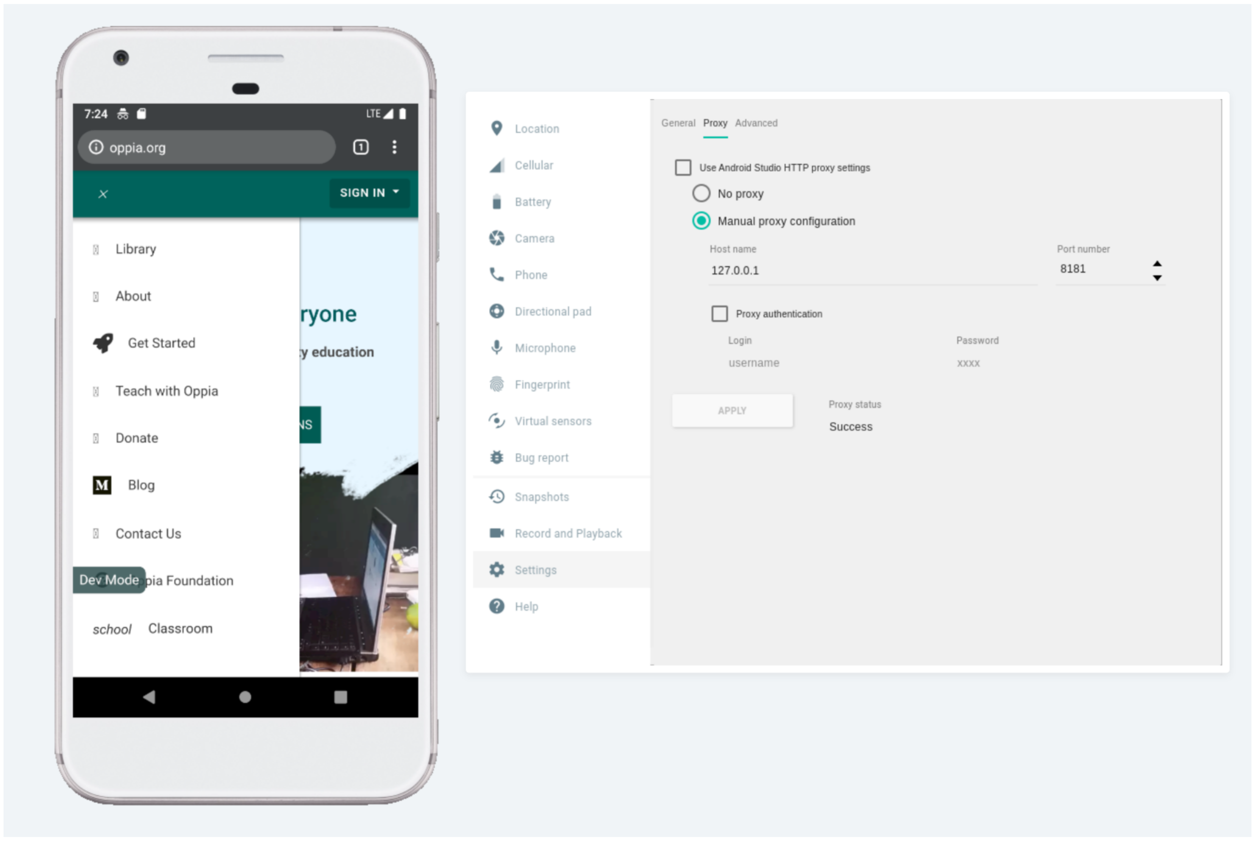
- I am able to run Android Instrumentation Tests (UiAutomator) using Bazel



```
farees@elementary-os:~/dev/bazel/testing-samples$ bazel test //ui/uiautomator/BasicSample:BasicSampleInstrumentationTest_23_x86 --config=headless
Another command (pid=10821) is running. Waiting for it to complete on the server (server_pid=10832)...
DEBUG: /home/farees/.cache/bazel/_bazel_farees/dc816a645c596c7d51c4ab154ddf671b/external/bazel_toolchains/rules/rbe_repo/version_check.bzl:68:14:
Current running Bazel is ahead of bazel-toolchains repo. Please update your pin to bazel-toolchains repo in your WORKSPACE file.
DEBUG: /home/farees/.cache/bazel/_bazel_farees/dc816a645c596c7d51c4ab154ddf671b/external/bazel_toolchains/rules/rbe_repo/checked_in.bzl:125:14: rbe_default not using checked in configs; Bazel version 4.0.0 was picked/selected but no checked in config was found in m
ap {"0.20.0": ["8.0.0"], "0.21.0": ["8.0.0"], "0.22.0": ["8.0.0", "9.0.0"], "0.23.0": ["8.0.0", "9.0.0"], "0.23.1": ["8.0.0", "9.0.0"], "0.23.2": ["9.0.0"], "0.24.0": ["9.0.0"], "0.24.1": ["9.0.0"], "0.25.0": ["9.0.0"], "0.25.1": ["9.0.0"], "0.25.2": ["9.0.0"], "0.
26.0": ["9.0.0"], "0.26.1": ["9.0.0"], "0.27.0": ["9.0.0"], "0.27.1": ["9.0.0"], "0.28.0": ["9.0.0"], "0.28.1": ["9.0.0"], "0.29.0": ["9.0.0"], "0.29.1": ["9.0.0", "10.0.0"], "1.0.0": ["9.0.0", "10.0.0"], "1.0.1": ["10.0.0"], "1.1.0": ["10.0.0"], "1.2.0": ["10.0.0"
], "1.2.1": ["10.0.0"], "2.0.0": ["10.0.0"], "2.1.0": ["10.0.0"]}
INFO: Analyzed target //ui/uiautomator/BasicSample:BasicSampleInstrumentationTest_23_x86 (0 packages loaded, 0 targets configured).
INFO: Found 1 test target...
Target //ui/uiautomator/BasicSample:BasicSampleInstrumentationTest_23_x86 up-to-date:
  bazel-bin/ui/uiautomator/BasicSample/BasicSampleInstrumentationTest_23_x86
INFO: Elapsed time: 57.238s, Critical Path: 0.04s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
//ui/uiautomator/BasicSample:BasicSampleInstrumentationTest_23_x86 (cached) PASSED in 76.5s

INFO: Build completed successfully, 1 total action
farees@elementary-os:~/dev/bazel/testing-samples$ 
```

- I am able to load the development server in the android emulator



Use Android Studio HTTP proxy settings
No proxy
Manual proxy configuration
Host name 127.0.0.1
Port number 8181
Proxy authentication
Login username
Password xxxx
Proxy status Success

Other summer obligations

I will give my full commitment to the project within the timeline. I might run into some college exams and assignments in between, yet I will be obligated to organization work.

Communication channels

I regularly use these communication channels and will respond within an hour

- Hangouts
- Gitter (via GitHub)
- e-mail
- Discord

Application to multiple orgs

I'm only applying to Oppia.

Project Details

Product Design

The users of this project are the developers of the Oppia Android team who would be able to run the tests on their systems.

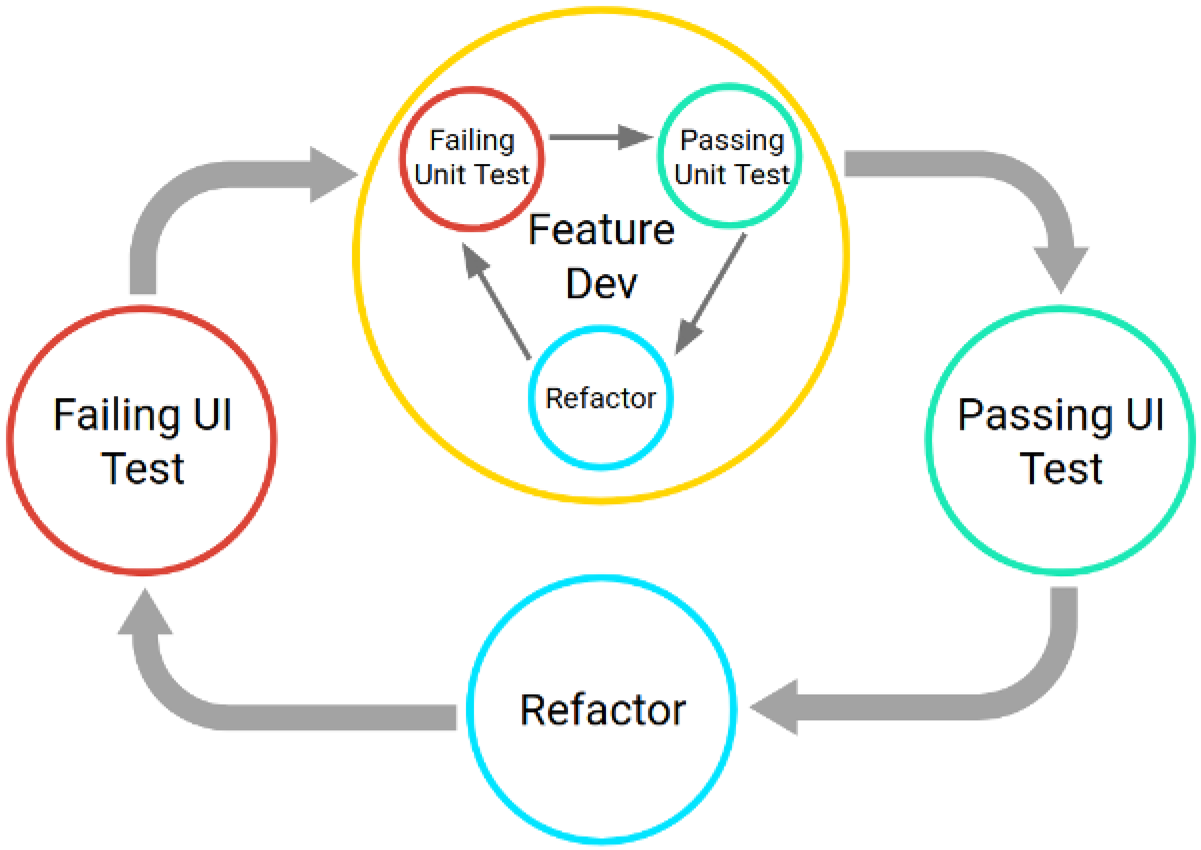
Why do we need to Test our App?

Let's suppose there is an app in which we need to check whether a particular feature/component of the app is working or not. we usually verify this by clicking around and verifying if it is working and if something doesn't work you can detect it easily and fix it and now as it is working perfectly and was verified then you don't touch it later. But when you extend your app by adding and updating features there might be something that breaks which were previously working and tested end then we need to test the whole app again by clicking around manually in this way it costs a lot of time and there might be some parts left in this procedure. This way the testing will be prone to mistakes and does not scale. To avoid this behavior of frequently testing the app manually after every update we write automated tests that manually operate the app to verify the feature in the various test cases.

The whole above procedure can be automated which gives a precise test result and also saves a huge amount of time. Testing your app is an integral part of the app development process. By running the tests against your app consistently after every update in your codebase, you can verify your app's correctness, functional behavior, and usability before releasing a feature or updating the codebase.

Testing also gives the following advantages:

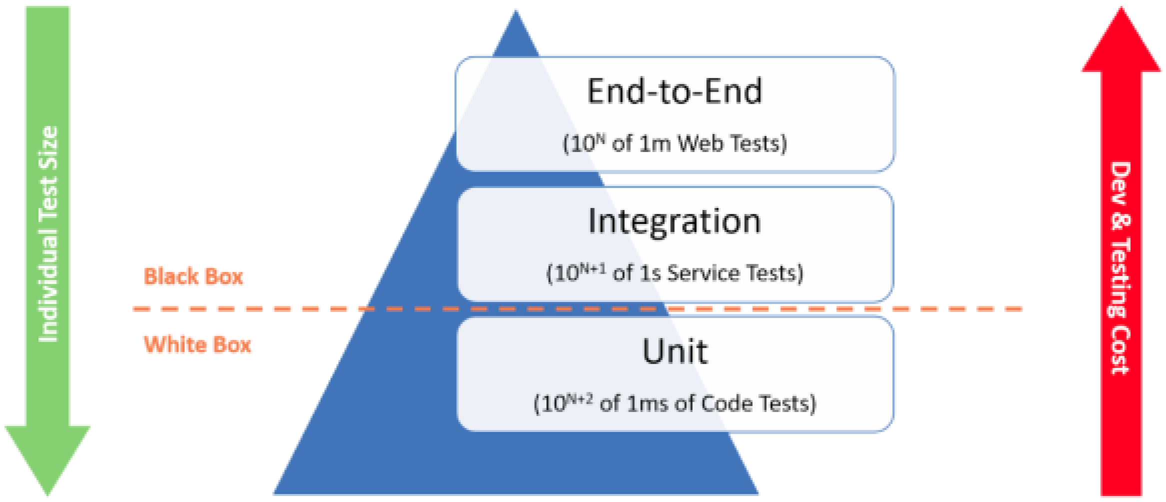
- Rapid feedback on failures
- Early Failure detection in the development cycle
- Safer Code refactoring, letting you optimize code without worrying about regressions.
- Stable Development Velocity, helping you minimize technical debt



Here the figure shows a Continuous testing and developing cycle. We see that when a Feature is developed it is tested across the app and the feature itself, In case the feature affects other parts of the app it is Refactored and fixed, and simultaneously when the feature is developed there were also tests written to test the feature across various test cases of the feature.

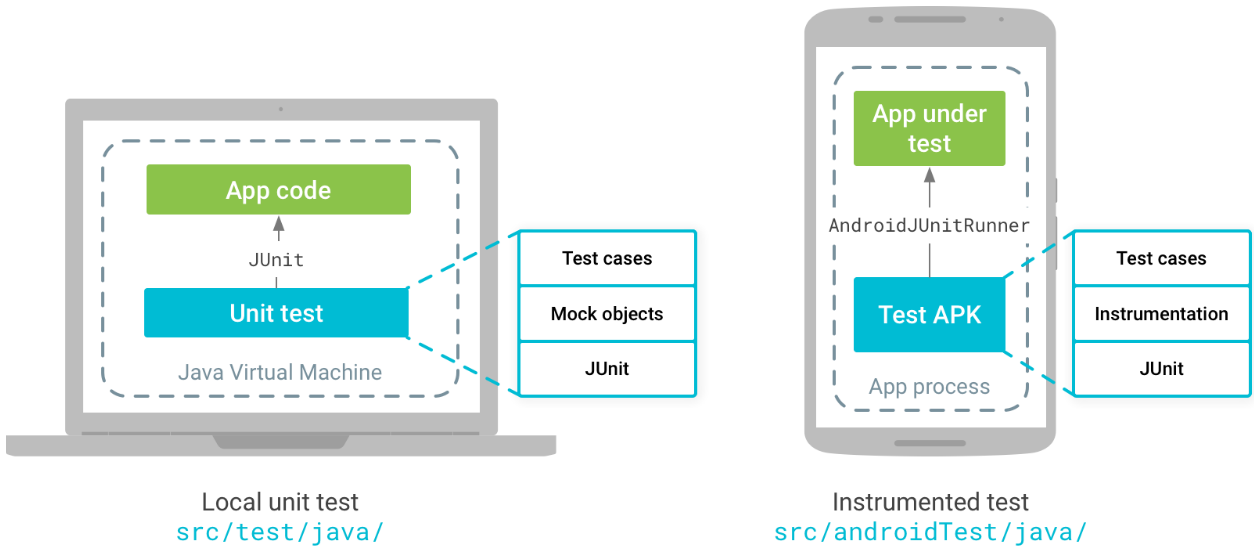
What are Android Instrumentation tests?

When we enter into Testing Android apps we come across several technical terms, which are divided into a test pyramid. The test pyramid is a tests dividing strategy, a way of thinking about how different kinds of automated tests should be used. In this division, the lower layer contains faster, cheaper, and isolated tests. At the top level of the pyramid, we will have slower, more expensive, and Integrated tests.

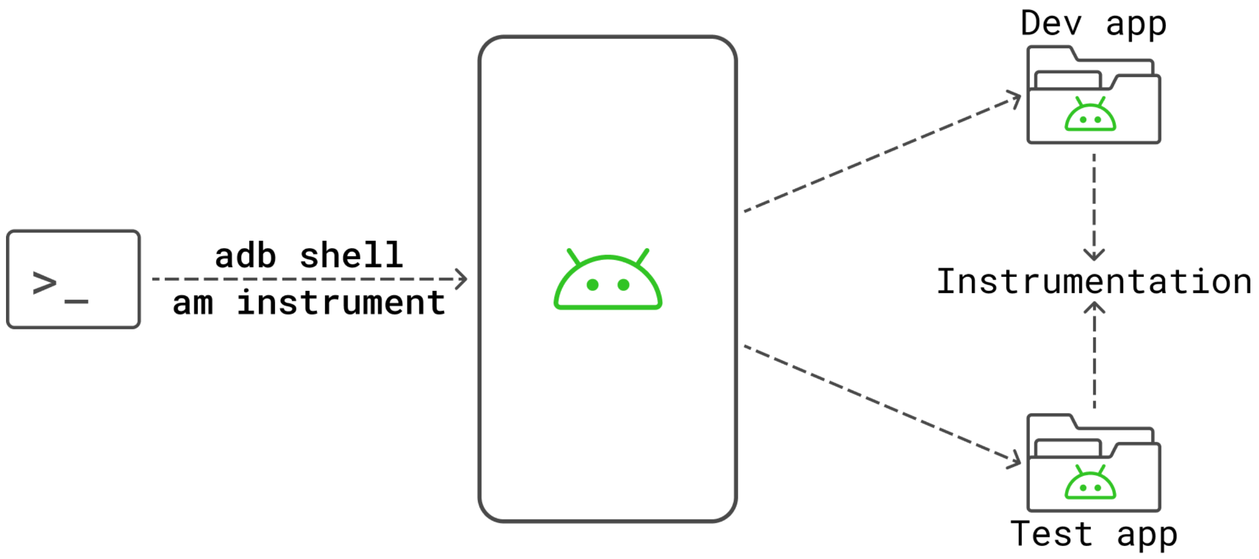


Here the white box represents local tests, the Black box represents the Instrumentation test (run on a real device or emulator)

This tests pyramid is divided into three parts:
Unit Tests: These are fast and cheap tests. This layer should have a greater quantity of tests than the upper layers. Usually tests logic. These make 70% of the testing in our app. Eg: Robolectric, JUnit, etc.
Integration Tests: To test how two components of our app work together i.e, to test the interaction between different components in the app. It is different from the integrated test(which requires our app to run in the emulator). These make 20% of the testing in our app. Eg: Espresso, UI Automator, etc.
End to End Tests: To test all the components of the app runs together including Network requests. These make 10% of the testing in our app. End-to-End tests. These make 10% of the testing in our app. Eg: UI Automator, Appmium, etc.



Instrumentation tests on Android are those that run on physical devices or emulators. These Instrumentation tests cover the Integrated tests and UI Tests of the **Test Pyramid**, These tests use AndroidJUnitRunner + AndroidJUnit4, Integrated tests with Espresso, and End-to-End tests with UI Automator. To be precise an instrumentation test provides a special test execution environment, where the targeted application process is restarted and initialized with the basic application context.



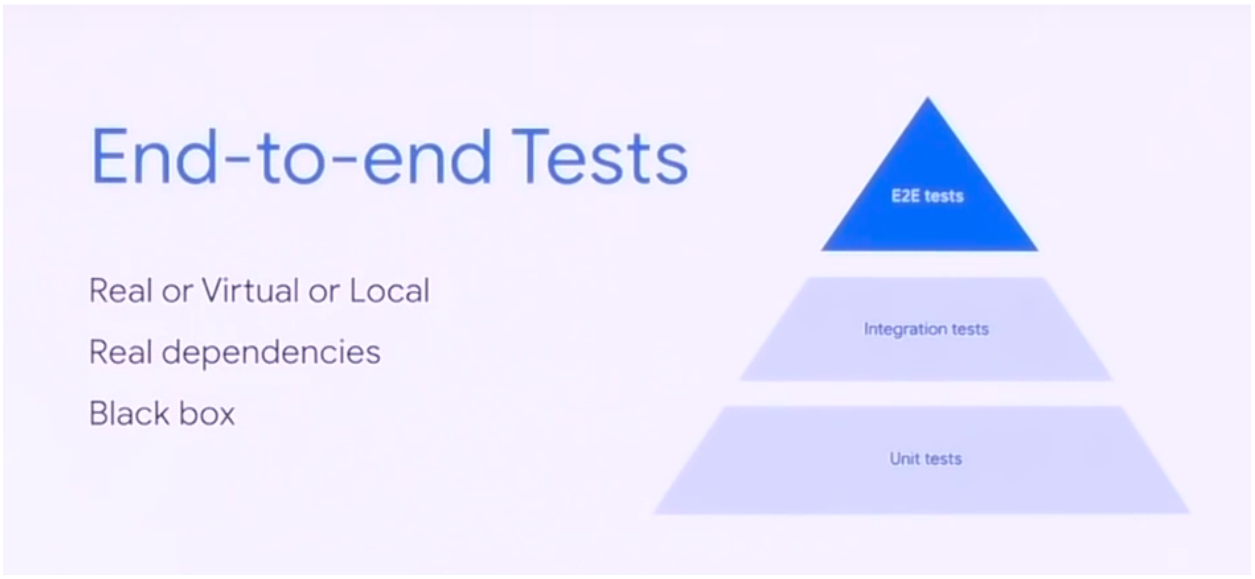
The above figure shows how an Android Instrumentation test is run i.e, by generating two applications, which will be installed in the device or emulator, which are the app we are developing and the test app that includes all test cases that need to be executed on the device or emulator. When both apps are installed on the device, the test app will interact with the developed app via Instrumentation API, which is a part of Android ROM. Here the am instrument is a command in ADB which is run internally to run the Instrumentation tests

Why do we need End-to-End tests?

The main purpose of End-to-End is to test the app from an end user's experience by simulating the real user scenario and validating the system under test and its components for integration and data integrity.

These tests play a major role in publishing the app. It gives confidence in the final application or a feature when it's finished and hence these tests should run on a real device or a virtual device to make sure that our code interacts with the Android environment as expected.

This application should be the same way as the final application and should test it in the same way that our users are going to interact with it (black-box testing)


End-to-end Tests

Real or Virtual or Local

Real dependencies

Black box

E2E tests
Integration tests
Unit tests

These tests whether the application is working flawlessly from start to finish based on the user's perspective.

What is End-to-End Testing and Why do we need a UI Automator instead of Espresso?

UiAutomator is a lightweight, easy-to-learn library, developed by Google to make things done fast when you don't want to spend lots of time on developing test code and its maintenance.

End-to-End testing is a software testing method that involves testing an application's workflow from beginning to end. This method aims to replicate real user scenarios so that the system can be validated for integration and data integrity. Essentially, the automated test goes through and verifies every operation the application can perform. Usually, End-to-End testing is executed after the Unit tests and Integrated tests as per the Test Pyramid.

End-to-End tests are slow as they require a huge amount of test cases to cover every aspect. These are slower, and harder to set up, but very valuable since they test almost all the test cases that are faced in real-time.

Both Espresso and UI Automator are used for Instrumentation Tests. Espresso is easy to use and fast test automation framework, but it has one important limitation -- we are not allowed to operate outside the app or the test application context Which makes it impossible to test automation tests for the following use cases

- Interacting with push notifications
- Accessing system settings
- Going back to previous activity from the current application context
- Navigation to other apps
- Using common intents (like camera, map, contacts, etc.)
- Using the application shortcuts

This limitation of Espresso can be removed by using UI Automator.

Espresso tests come under the **Integration tests** and it is not suitable for End-to-End tests due to its limitations, UI Automator comes under **End to End Tests**, UI Automator is perfect for End-to-End as it is stable and easy to maintain highly time-sensitive operations, unlike Espresso.

Ui Automator is a lightweight, easy-to-learn library, developed by Google to make things done fast when you don't want to spend lots of time on developing test code and its maintenance.

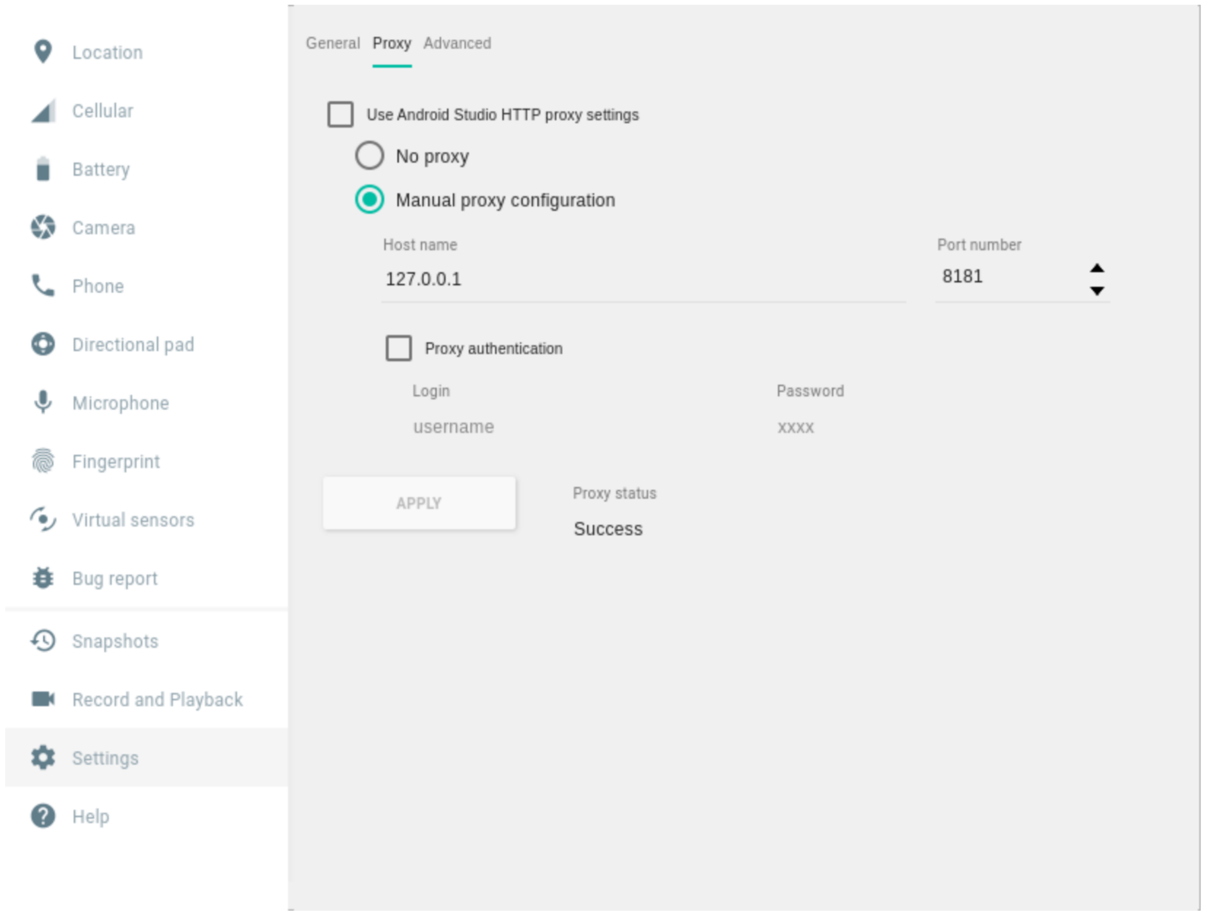
With UI Automator it is possible to access outside the app or the test application context. UI Automator is a UI testing framework suitable for cross-app functional UI testing across systems and installed apps.

How to Interact the app with the local development server of oppia to load the test Data?

First, we need to set up the local development server with oppia such that it provides an API endpoint in the localhost, we need to set up a developer-only functionality for this endpoint that can prepopulate tests topics, stories, exploration, revision cards, skills, and questions in the local datastore.

The only way to communicate with two different projects is with the API and the network requests. The backend will be providing an API Endpoint and the Android app responds to these endpoints through Network Requests.

After setting up the local Development Server, we need to set up Android Emulator networking. The emulator provides networking capabilities that can be used to set up complex modeling and testing environments for the app. By setting up the proxy settings in the emulator we can test the Networking in the app without the hosted development server so that the End-to-End tests are not flaky and faster testing as the networking using the local development server is not dependent on the internet.



Use Android Studio HTTP proxy settings

No proxy

[✓ Manual proxy configuration]

Host name: 127.0.0.1

Port number: 8181

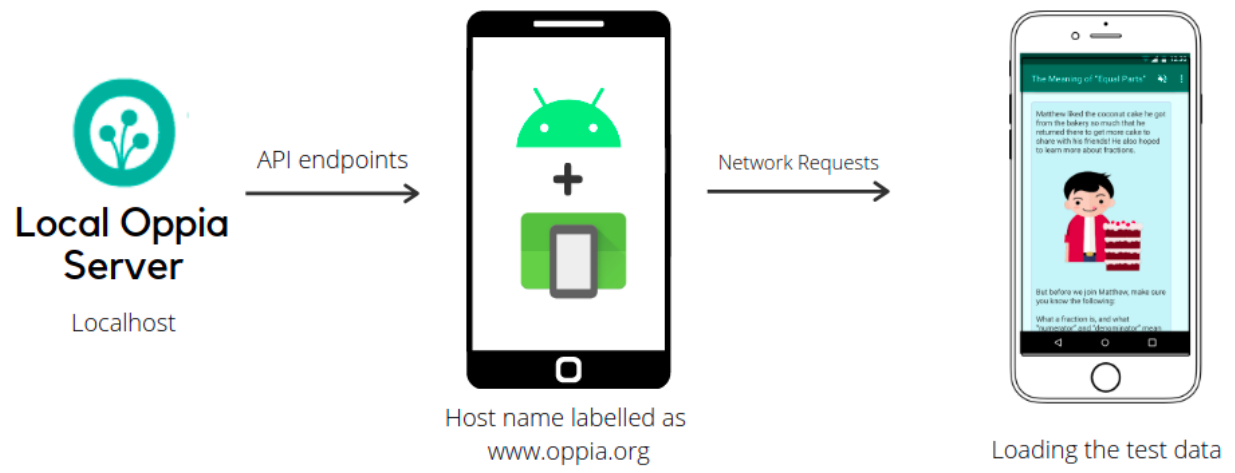
Proxy authentication

Login: username

Password: xxxx

Proxy status: Success

In the proxy settings (from the above figure) we can add a hostname used in the app to the port number of the server to Interact with the app, This way we can load the test data in the app and verify the Network Requests in the app.



End Result of the Finished Project

The End Result of the Project provides a developer-only local API from the oppia web project that is set up using a command

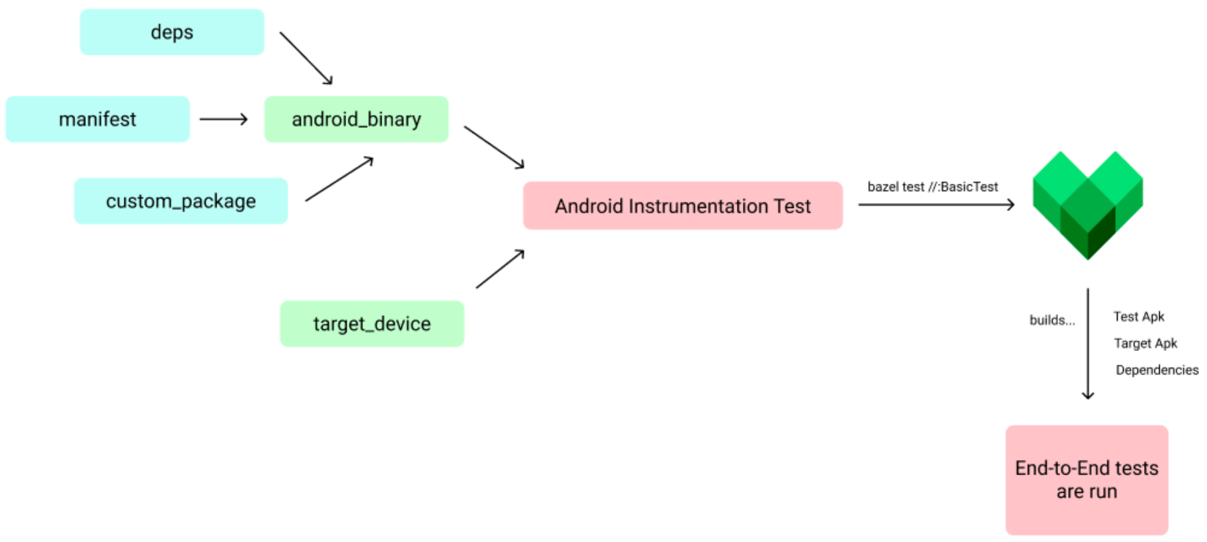
```
python -m start.py --disable_host_checking
```

to start the server and call the developer endpoint which initiates the test data that includes topics, stories, chapters, explorations, revision cards, skills, and questions using the curl command i.e,

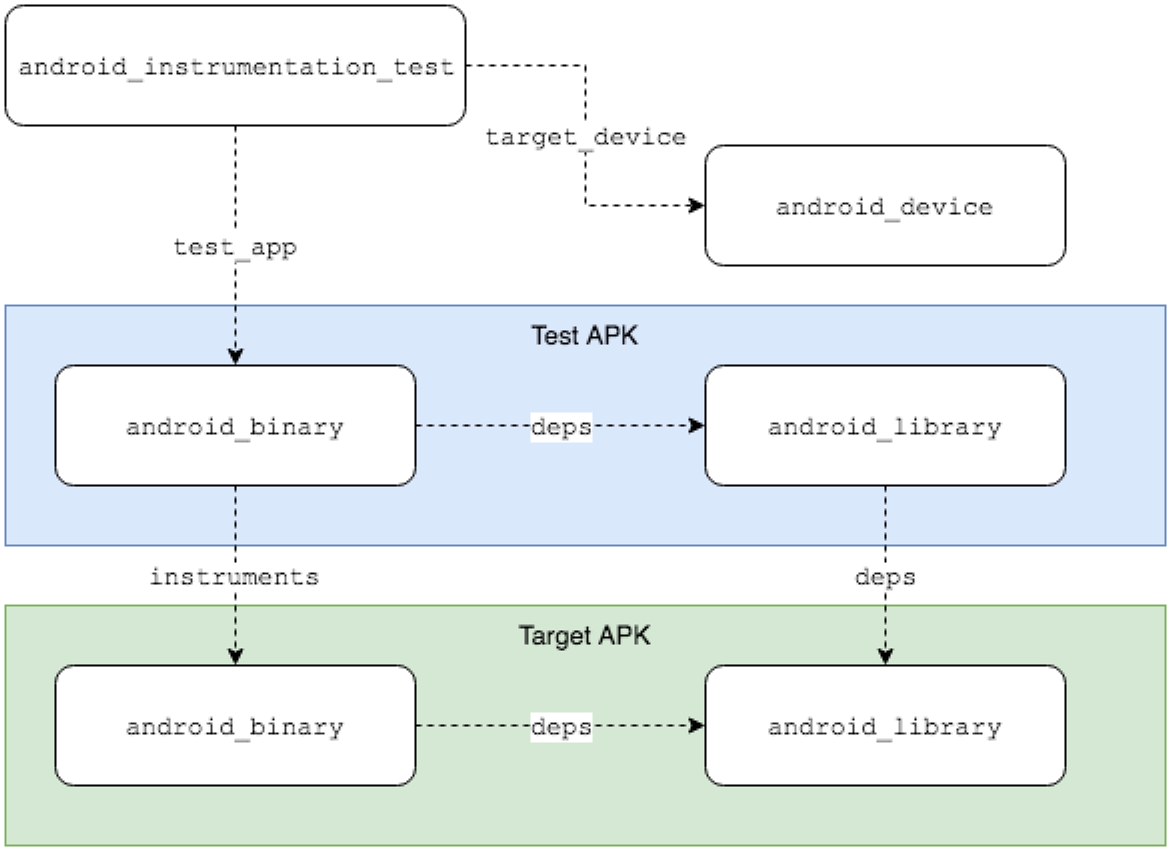
```
curl http://localhost:8181/initiate_test_data
```

In the app, the UI Automator will be used and set up with **Bazel** to run the End-to-End tests. **Bazel** is a build and test tool which is cross-platform and builds Java, C++, Android, IOS, Go, Kotlin, and a wide range of other language platforms. It speeds up your Build and test progress through distributed caching and parallel execution. It is Extensible to your needs i.e, easily supports new languages and platforms. Configuration files (BUILD and WORKSPACE) in bazel are much more structured than Gradle. Bazel is a cross-platform build tool the whole application that consists of different languages or frameworks can be Build and tested altogether.

Test targets will be added for each test suite to be able to run in bazel. Targets in bazel are similar to declaring variables i.e, simply variable to a Bazel Rule, these targets are declared to run or build for one or more than one file along with the dependencies that are required for the target called as deps.



To run the Instrumentation tests in Bazel for an existing app we require an android_binary of the test suite and create an android_instrumentation_test with a target_device and when we run the tests it Generates a Test apk, APK under the test instructions and their transitive dependencies. This Creates and starts an emulator, installs the app, Runs the tests the same as how it works in Gradle, shuts down the emulator, and finally sends a report on the tests.



The above figure shows the dependency graph of an Android Instrumentation test using bazel. On how the test APK and the Target APK are built when Instrumentation tests are run.

Finally, the End-to-End tests will be covering the Complete user instruction for the Explorations, Exploration is an amazing feature of Oppia which is Interactive and enjoyable for the students to learn at their own pace. These Explorations provide feedback for every answer so that the student can have a better understanding of the lessons and have a feeling of interacting with a teacher. This project is a beginning point and a reference for lots of End-to-End tests in the future for Explorations (if updated) and other parts of the app. End-to-End tests in this will also help to find bugs around the Explorations and will ignore them till the bug is fixed.

Technical Design

Architectural Overview

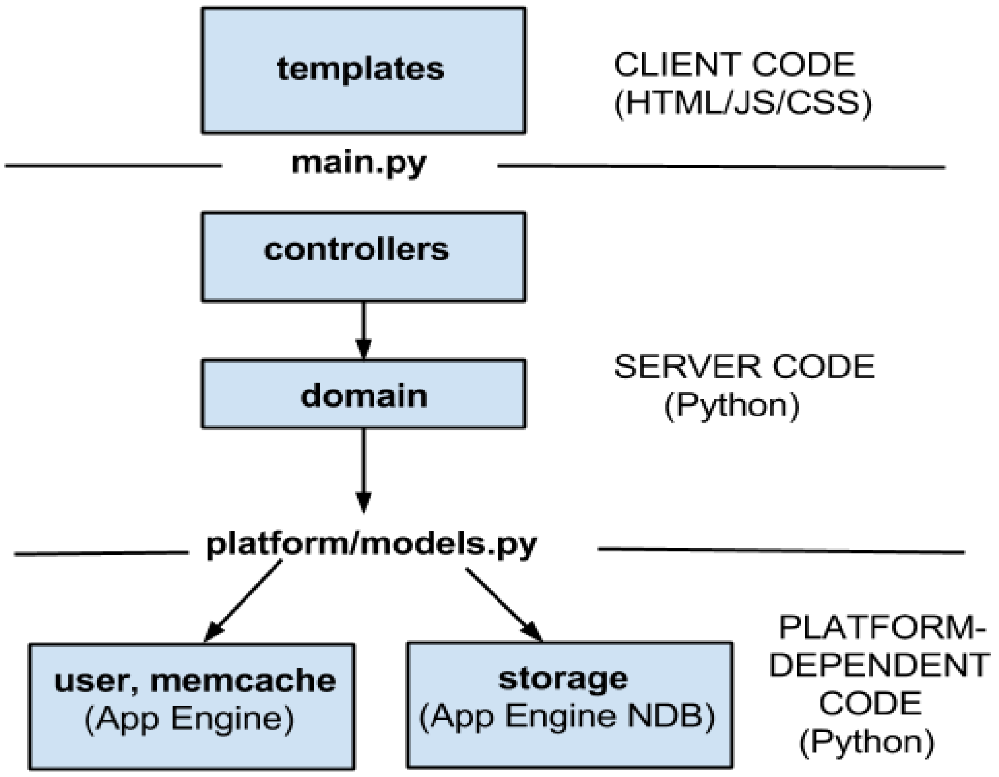
The whole project in short involves creating API endpoints and respective controllers, Network Requests, Setting up the UI Automator, Adding End-to-End tests.

Dividing the Architectural overview between the two repositories (Oppia-web and Oppia-Android) Explaining which part of the codebase and the usage, what code goes where and the reason

Oppia-Web

Codebase of Oppia-web:
Most of the functionality goes under the core directory with the following modules based on particular features and purposes as follows:

- Controllers
- Domain
- Platform
- Storage
- Templates
- Tests



Backend
A request to the Oppia server by performing some action (like clicking a button) that causes some JavaScript code to issue a POST request. This request is made to a particular URL, which main.py matches to a handler in the core/controllers directory.

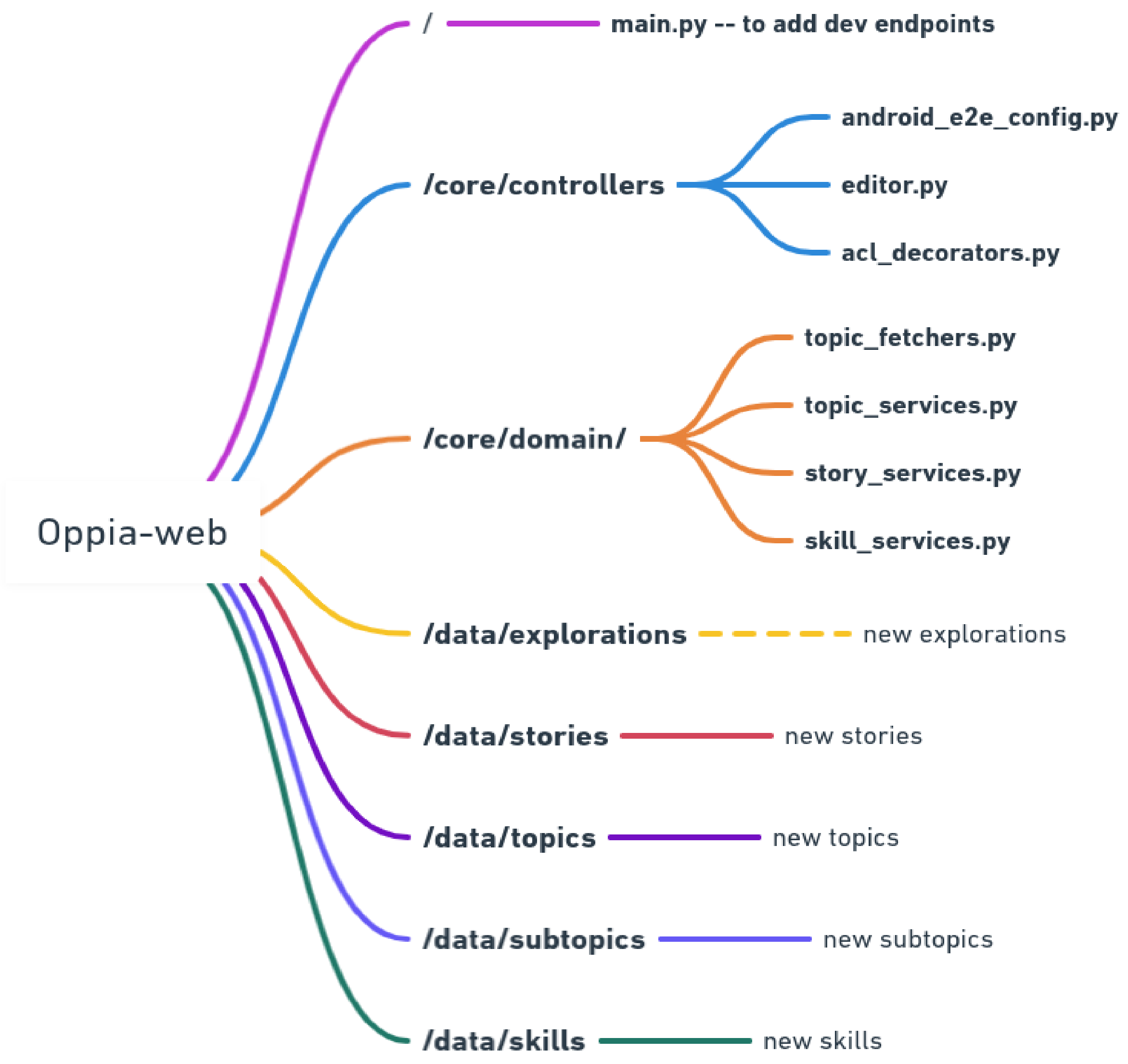
/core/controllers -- [Directory] Urls from the main.py matches with the Handlers in the Controllers directory. Controllers are meant to be a thin layer that understands and validates the requests, These Controllers/Handlers then call a method from the core/domain.

/core/domain -- [Directory] Performs the computation/queries or changes the state of data on the server for the respective handlers. Here we need *_service.py as it comprises the functions based on Data.

Frontend
The developer version of the frontend code is contained in core/templates. The frontend code contains the following sub-directories:

- *components*: This provides certain reusable components, structured as Angular directives, which are used in one or more pages.

- *css*: Site-wide CSS. There may be other CSS blocks within individual HTML files.
- *domain*: The logic layer for the frontend. Contains domain object factories and core services.
- *expressions*: Code to parse and evaluate expressions (which may use parameters).
- *pages*: Each folder in this directory represents an individual page of the site. Some of these pages are quite simple (such as /about) whereas others are fairly complex (such as /exploration_editor). They contain the HTML templates for the page, as well as any page-specific services or directives.
- *services*: JavaScript services that handle the embedding of explorations.



Main Directory

/main.py - [File] Register new urls and their respective Controllers/Handlers through get_redirect_route function. To add new Web URLs that redirect web page URLs to the page/ handler desired in the request

feconf.py - [File] Contains various constants that are referred to by other backend files in the app. To add new constants for the new endpoints.

Controllers Module

/core/controllers/android_e2e_config.py - New controller for the following classes/handlers

- InitializeAndroidTestDataHandler -- new class/handler which updates the local assets to the datastore
- AndroidTestImageHandler -- new class/handler which can access the images from the local dev server.

/core/controllers/topic_editor.py -

- TopicFileDownloader → download topic from the datastore.
- SubtopicFileDownloader → download subtopic from the datastore.

/core/controllers/story_editor.py -

- StoryFileDownloader → download story from the datastore.

/core/controllers/skill_editor.py -

- SkillFileDownloader → download skills from the datastore.

/core/controllers/acl_decorators.py - To modify can_download_explorations to support all other structures.

Domain Module

/core/domain/topic_services.py - Introducing support to export zip files from the topic and subtopic and vice versa

/core/domain/topic_fetchers.py - Introducing get_subtopic_by_id function to fetch a subtopic.

/core/domain/story_services.py - Introducing support to export zip files from a Story and vice versa

/core/domain/skills_services.py - Introducing support to export zip files from a Skill and vice versa

/core/domain/story_services.py - Introducing support to export zip files from a story and vice versa

Data Directory

/data/explorations - [Directory] Contains sample explorations that are bundled with the Oppia distribution.

/data/topics - [new Directory] To add sample topics that are extracted or fetched from the datastore

/data/subtopics - [new Directory] To add sample subtopics that are extracted or fetched from the datastore

/data/stories - [new Directory] To add sample stories that are extracted or fetched from the datastore

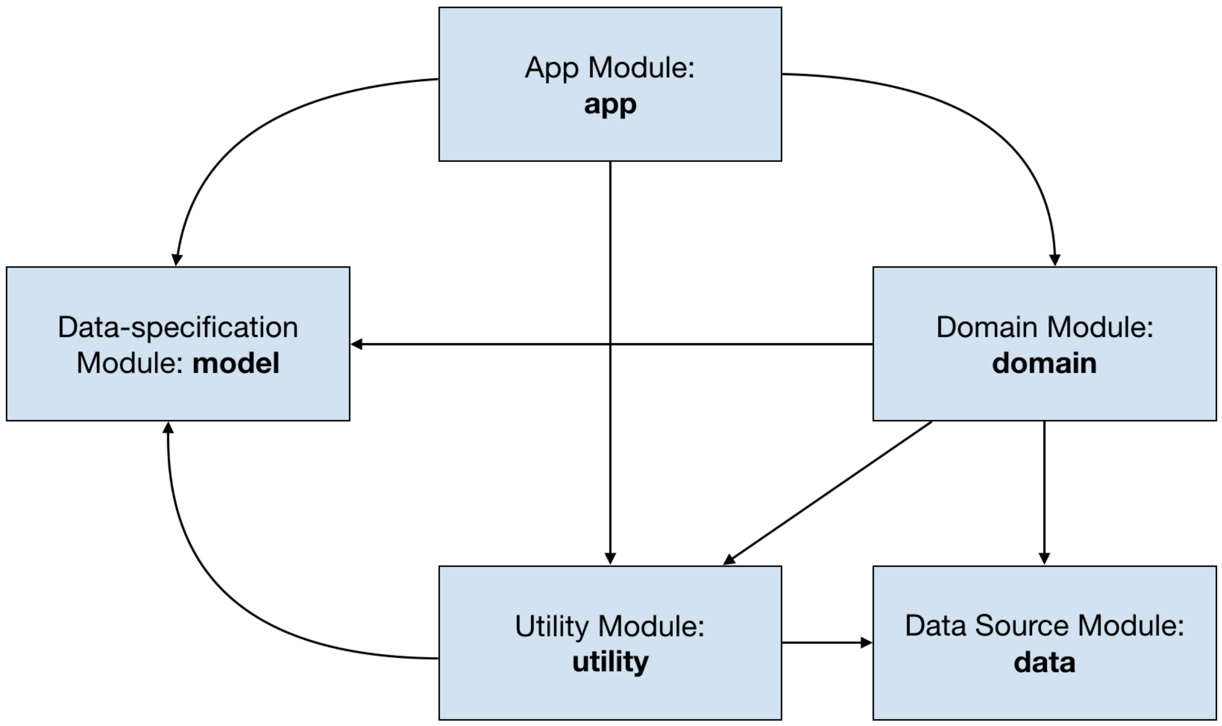
/data/skills - [new Directory] To add sample skills that are extracted or fetched from the datastore

Oppia-Android

Oppia's Android codebase has five modules based on particular features or purposes.

- App
- Domain
- Model
- Utility
- Data

The following diagram represents co dependency of each module with other modules.



Data - This module provides data to the application by fetching data from the Oppia backend. It has two subdirectories

- Backend -- APIs and models which are needed to make a data request to the Oppia backend, and convert that response to appropriate models.
- Persistence -- Information pertaining to the offline store is saved here using PersistenceCacheStore

App - This module contains all the activities and fragments, as well as the view, view model, and presenter layers. It also contains Robolectric test cases and integration/hermetic end-to-end tests using Espresso.

Domain - This module contains the business logic of the application, including both frontend controller and business service logic. It is a Java/Kotlin library without Android components, and it is unit-tested using raw JUnit tests.

Model - This library contains all protos used in the app. It only contains data definitions, so no tests are included.

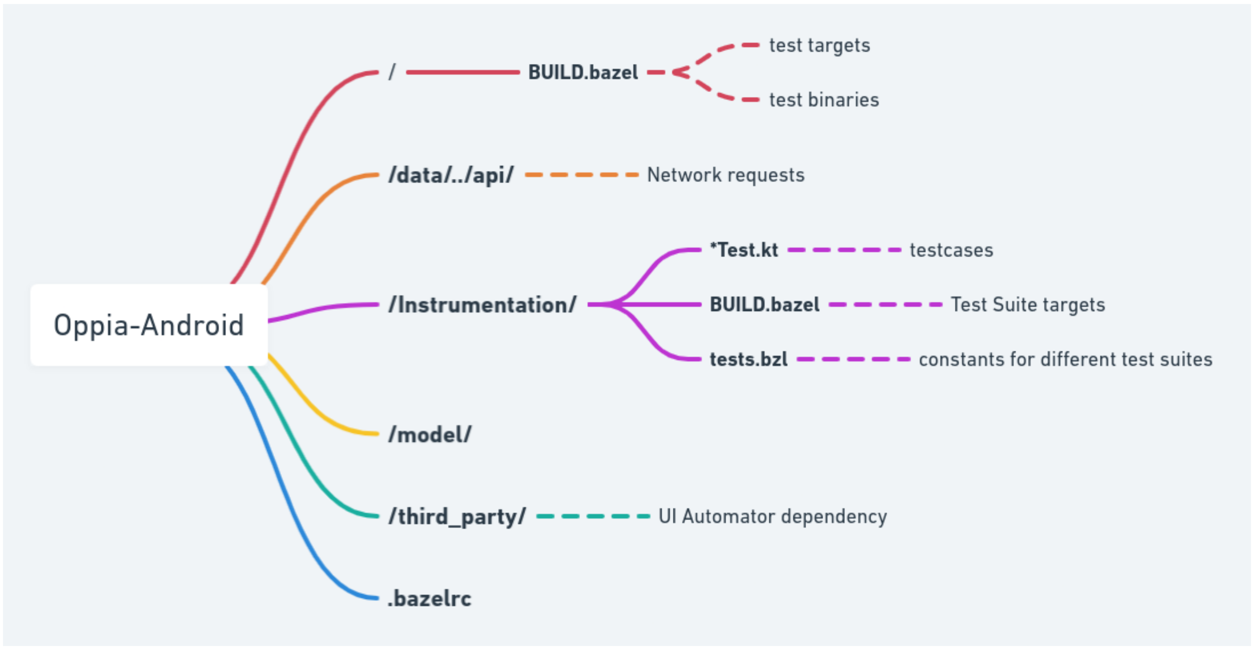
Utility - This is a Java/Kotlin module without Android dependencies. It contains utilities that all other modules may depend on. It also includes JUnit test cases.

The following are different from the modules Introduced for bazel

instrumentation - [Directory] New top-level Directory to be Introduced for the End-to-End tests.

BUILD.bazel - [File] Corresponds to being accessible to all Oppia targets. This should be used for production APIs & modules that may be used both in production targets and in tests.

third_party - [Directory] This package contains all third-party dependencies that the project depends on.



Main Directory

Utility Module

util/parser/ImageParsingModule.kt - [File] Provides image-extraction URL dependencies.

To add a test GCS prefix to android_test_images endpoint to act as a Mock GCS that fetches images from the local server.

/.bazelrc - [File] Bazel looks for optional configuration files in the following locations, in the order shown below. The options are interpreted in this order, so options in later files can override a value from an earlier file if a conflict arises. All options that control which of these files are loaded are startup options, which means they must occur after bazel and before the command (build, test, etc).

To Add configurations for instrumentation tests to run on local emulator or different options.

/BUILD.bazel - [File] Corresponds to being accessible to all Oppia targets. This should be used for production APIs & modules that may be used both in production targets and in tests.

To add android_binary and android_instrumentation_test of the Bazel Android Rules related to the End-to-End tests or Instrumentation tests.

/instrumentation/ - A newly Introduced top-level Directory for Instrumentation tests particularly for End-to-End tests. This consists of Tests Suites (*Test.kt), BUILD.bazel for kt_android_library based on the imports, tests.bzl a constant file similar to /third_party/versions.bzl to dynamically run particular Instrumentation tests.

instrumentation/player/exploration/ExplorationTests → Tests that include Downloading and Playing through an exploration user flows.

/third_party/version.bzl - To add UI Automator Dependency

Implementation Approach

Dividing the project into the further milestone or parts such that each of the following are not dependent on others:

1. Downloading Non-Exploration structures from the datastore -- to download the topics using a new endpoint/controller similar to /createhandler/download/<exploration-id> .
2. Create android specific custom structures -- To create sample explorations in order to test the implemented features of android in one exploration.
3. Initiating the Test Data -- To save the local topics/explorations into the local datastore.
4. Setup UI Automator with Bazel -- To setup UI automator in order to run the UI Automator tests.
5. Testing with UI Automator -- Writing End-to-End tests using UI Automator for End-To-End testing.

Downloading Non-Exploration structures from the datastore

This part blocks the Initiating Test data and custom explorations part as we don't have any support for downloading Topics, Skills, Stories.

Note: Each of Explorations, Stories, Topics, Subtopics, Skills are represented below as **Structure** or ***.**

First we need to create a new endpoints similar to /createhandler/download/<exploration_id> for Topics, Skills and Stories.

For better readability we need to rename the existing endpoint for explorations downloads as follows

/createhandler/download/<exploration_id> → /createhandler/download_exploration/<exploration_id>

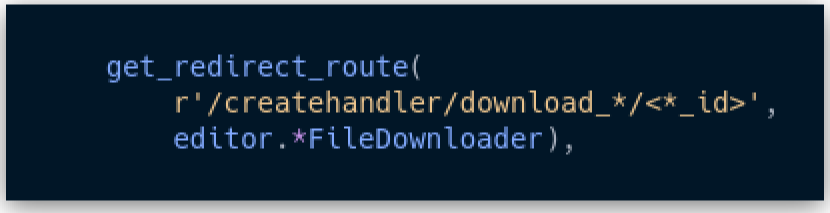
To download each of the Explorations, Stores, Skills & Topics (let's call it a **Structure**). Here we need to introduce 4 endpoints (Note that: Exploration is renamed here)

- /createhandler/download_explorations/<exploration_id>
- /topic_editor_handler/download_topics/<topic_id>
- /topic_editor_handler/download_subtopic/<subtopic_id>
- /story_editor_handler/download_stories/<story_id>
- /skill_editor_handler/download_skills/<skill_id>

These four endpoints redirect to a Handler from the editor controller (i.e, core.controllers.editor.py) as are added as follows

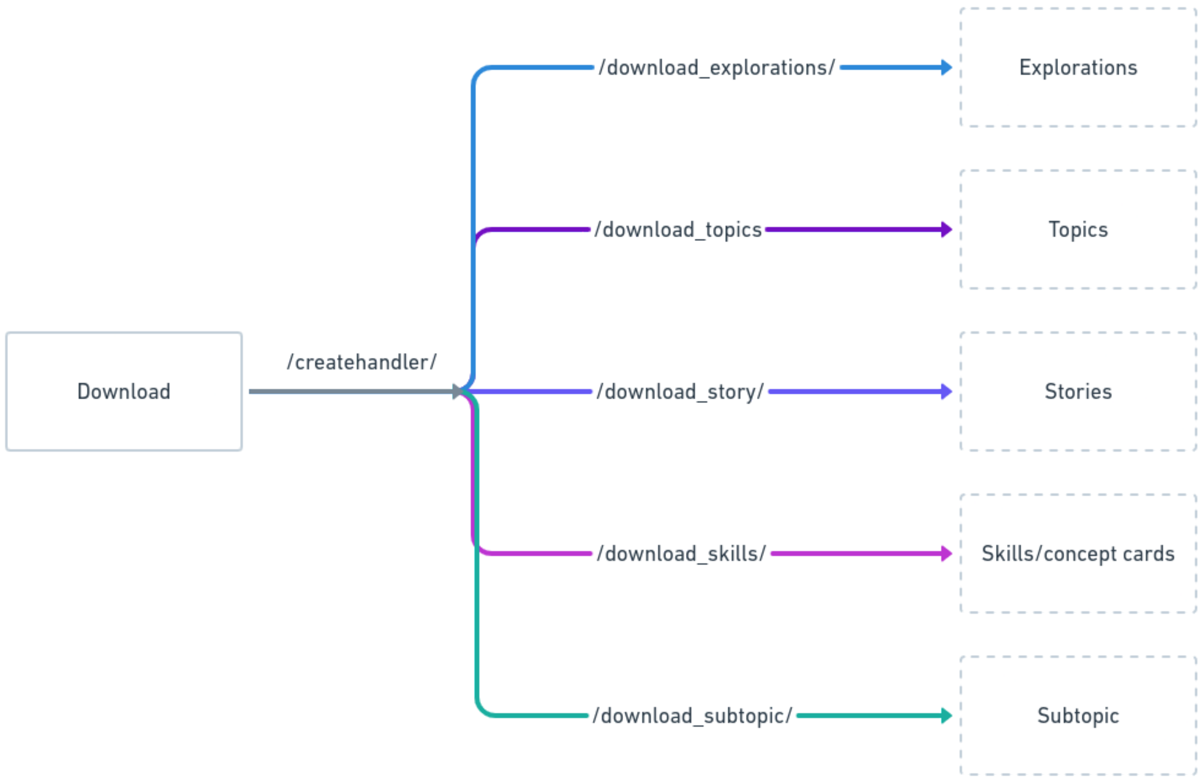
To differentiate both endpoints and better readability.

After adding the new endpoints, these endpoints redirects to their corresponding downloaders from the editor controller core.controllers.editor.py



```
get_redirect_route(
    r'/createhandler/download_*/<*_id>',
    editor.*FileDownloader),
```

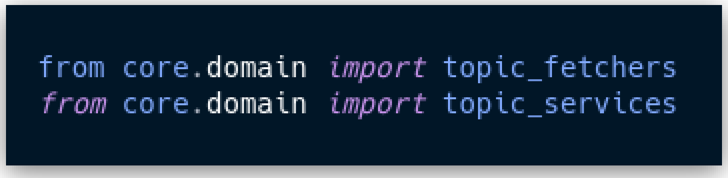
In the above snippet the “ * “ represents each of the exploration, topic, subtopic, story and skill



Create a new classes i.e, the downloader to which the above endpoint are redirected to Corresponding editors as follows

- download_explorations → core/controllers/editor.py
- download_topics → core/controllers/topic_editor.py
- download_subtopics → core/controllers/topic_editor.py
- download_stories → core/controllers/story_editor.py
- download_skills → core/controllers/skill_editor.py

To get the info for each structure we need to import their respective *_fetchers and *_services Example: fetchers and services for Topics



```
from core.domain import topic_fetchers
from core.domain import topic_services
```

As mentioned above we don't have any downloaders for Topics, Subtopics, Stories, Skills.

Downloading Topics

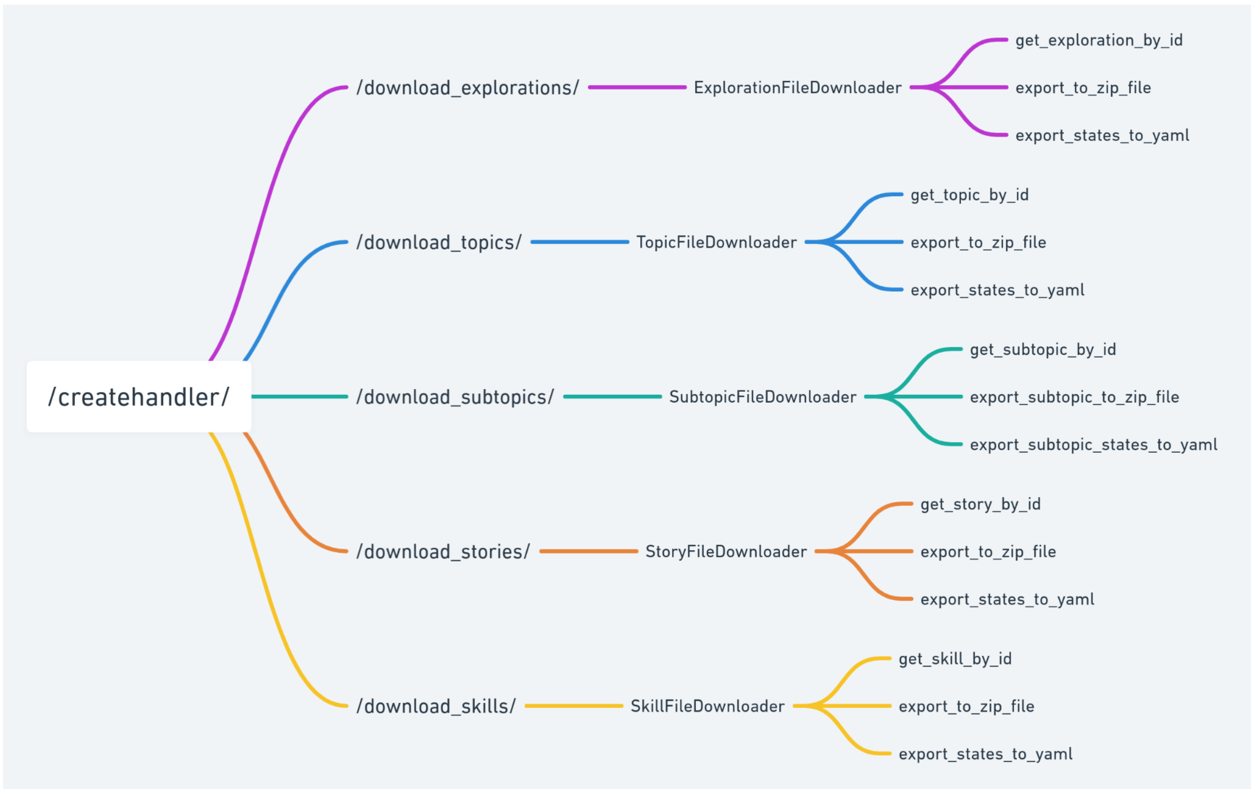
Introduce a new class which is redirected by the /createhandler/download_topics/<topic_id> endpoint i.e, TopicFileDownloader. The main purpose of this class is to download the topic from datastore as a zip file.

The topic is fetched using the get_topic_by_id function from core.domain.topic_fetchers.py this topic (object) is used to get the topic name and version in order to name the zip file. Finally the zip file is fetched from the core.domain.topic_services.py where a new function is introduced i.e, export_to_zip_file and downloaded locally.

The export_to_zip_file at a high level gets the topic from the ID, converts it into a yaml file and adds the local assets into an images folder and then all these are added into a zip file.

Here to convert the topic to a yaml file in case if the the output_file in TopicFileDownloader, there is a new function to be introduced in the core.domain.topic_services.py i.e, export_states_to_yaml which converts the topic into a dictionary of the topic whose keys are state names and values are yaml strings.

This process is the same for others all other structures i.e, Stories, Subtopics, Skills. I.e, as follows



The above mentioned functions and class have the same logic or implementation for each of them.

The following functions that are mentioned above are not available and need to be introduced in this project which will be used in the respective FileDownloader class

- **Topics:**
  - export_to_zip_file (core.domain.topic_services.py)
  - export_states_to_yaml (core.domain.topic_services.py)
- **Subtopics:**
  - get_subtopic_by_id (core.domain.topic_fetchers.py)
  - export_subtopic_to_zip_file (core.domain.topic_services.py)
  - export_subtopic_states_to_yaml (core.domain.topic_services.py)
- **Stories:**
  - export_to_zip_file (core.domain.story_services.py)
  - export_states_to_yaml (core.domain.story_services.py)
- **Skills:**
  - export_to_zip_file (core.domain.skill_services.py)
  - export_states_to_yaml (core.domain.skill_services.py)

export_to_zip_file

Returns the zip archive of the structure
Converts the structure to yaml (using to_yaml from core.domain.collection_domain.py) separates the images into a different directory (named: images or assets) and finally returns the zip.

Attribute name	Description
structure_id	The id of the structure to export.
version	(Optional -- default is None) Indicates which version of the structure to export, else the latest version is exported

export_states_to_yaml

Returns a dictionary of state names and yaml strings to save as a json.
To get the json if the output file of the downloader class is of json format
Keys - state names
Values - yaml strings wrapped with width

Attribute name	Description
structure_id	The id of the structure to export.
version	(Optional -- default is None) Indicates which version of the structure to export, else the latest version is exported
width	(Optional -- default is 80) Width for the yaml representation.

get_subtopic_by_id

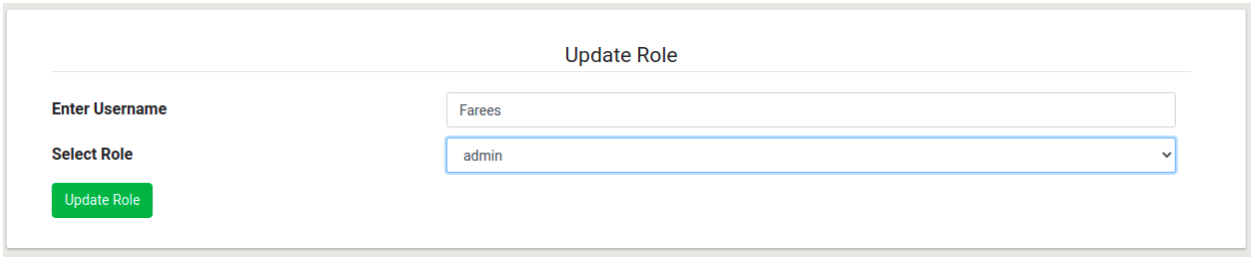
Returns a domain object representing a subtopic.
Gets the subtopic model using the ID.

Attribute name	Description
subtopic_id	ID of the subtopic
strict	(Optional -- default is True) Whether to fail noisily if no subtopic with the given id exists in the datastore.
version	(Optional -- default is None) The version number of the topic to be retrieved. If it is None, the latest version will be retrieved.

Create android specific custom structures

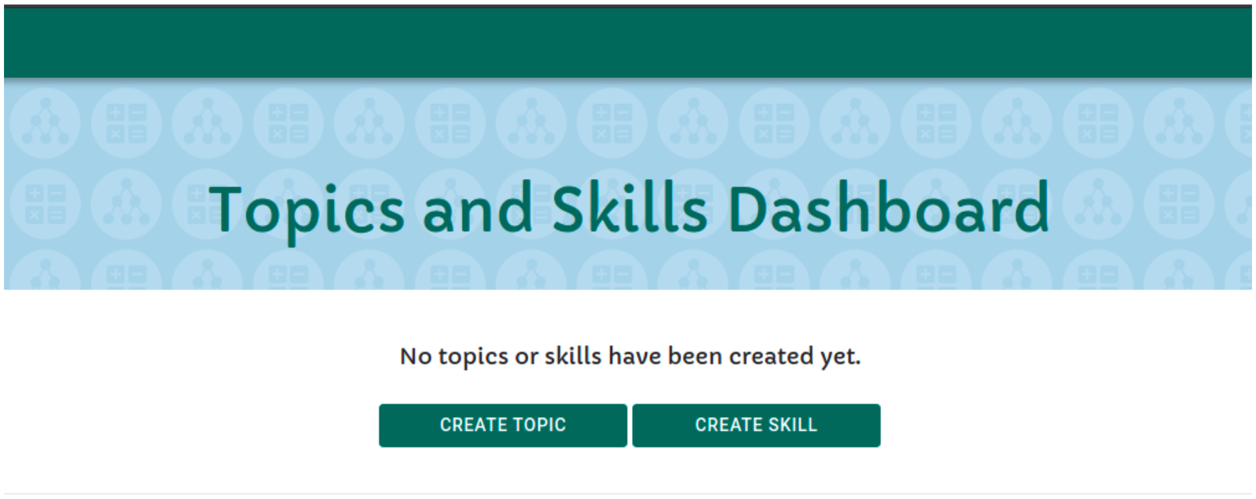
To create a new custom exploration, start the development server after following all the installation steps, by using the command `python -m scripts --save_datastore` (only to preserve the exploration and other contents in the local datastore)

Explorations can be created using the dev server itself through GUI. To create a new Topics, Skills, and Explorations first we need to login and assign a admin role using the following Endpoint --`/admin#/roles`


Update Role

Enter Username	Farees
Select Role	admin

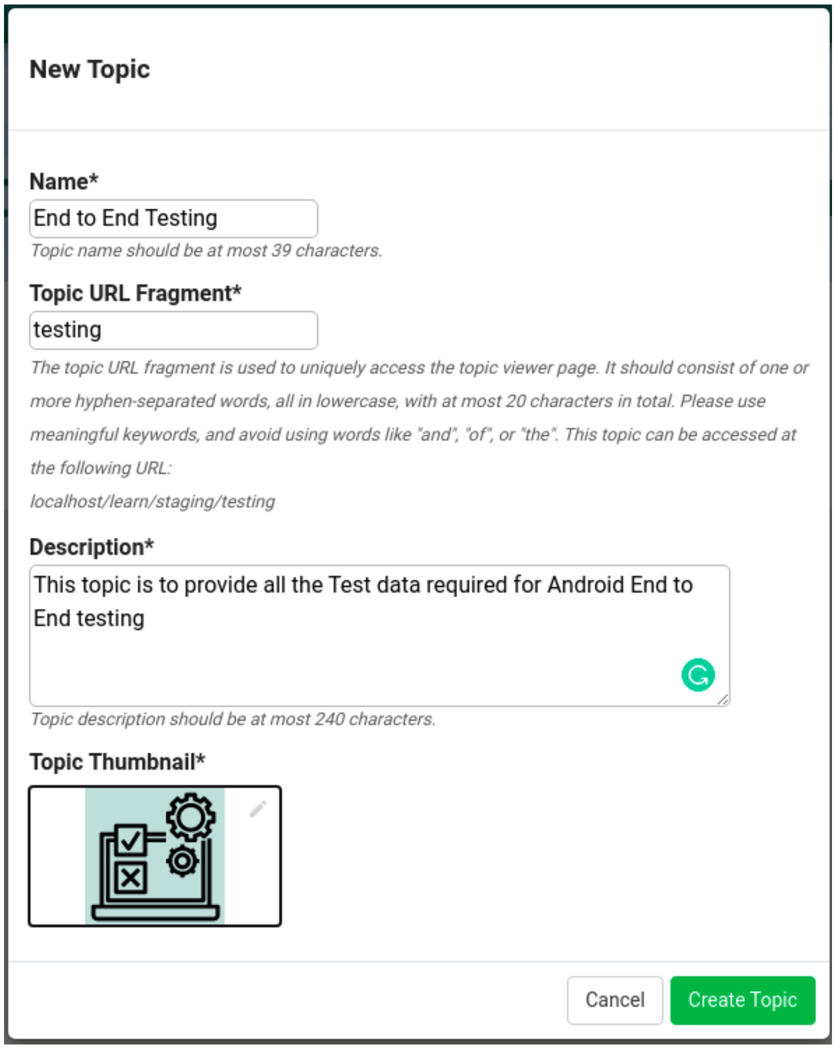
Then open the endpoint /topics-and-skills-dashboard to create new topics, concept cards, revision cards, etc.,


Topics and Skills Dashboard

No topics or skills have been created yet.

After opening the above endpoint click on “CREATE TOPIC” fill the

- Name
- Topic URL Fragment
- Description
- Topic Thumbnail


New Topic

Name*

End to End Testing

Topic name should be at most 39 characters.

Topic URL Fragment*

testing

The topic URL fragment is used to uniquely access the topic viewer page. It should consist of one or more hyphen-separated words, all in lowercase, with at most 20 characters in total. Please use meaningful keywords, and avoid using words like "and", "of", or "the". This topic can be accessed at the following URL:

localhost/learn/staging/testing

Description*

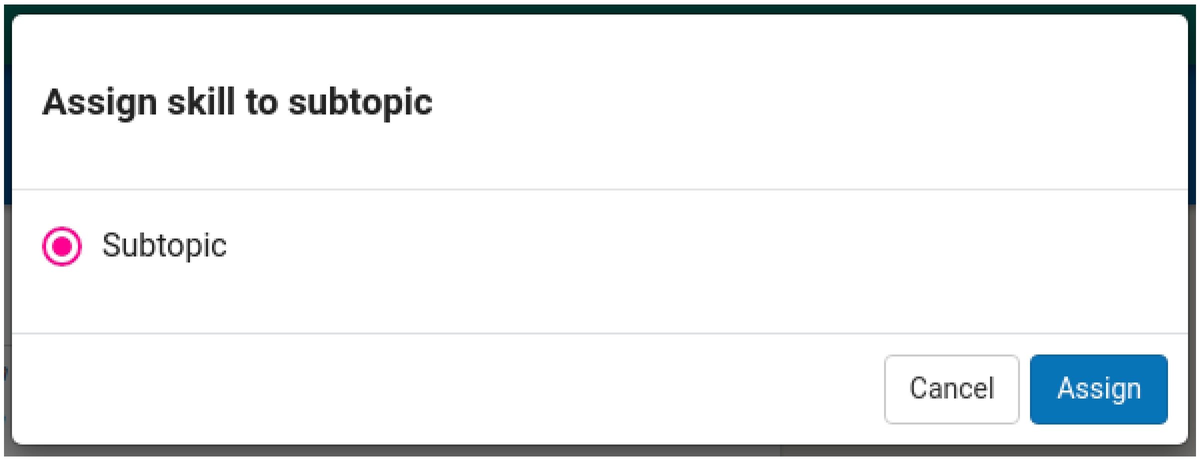
This topic is to provide all the Test data required for Android End to End testing

Topic description should be at most 240 characters.

Topic Thumbnail*

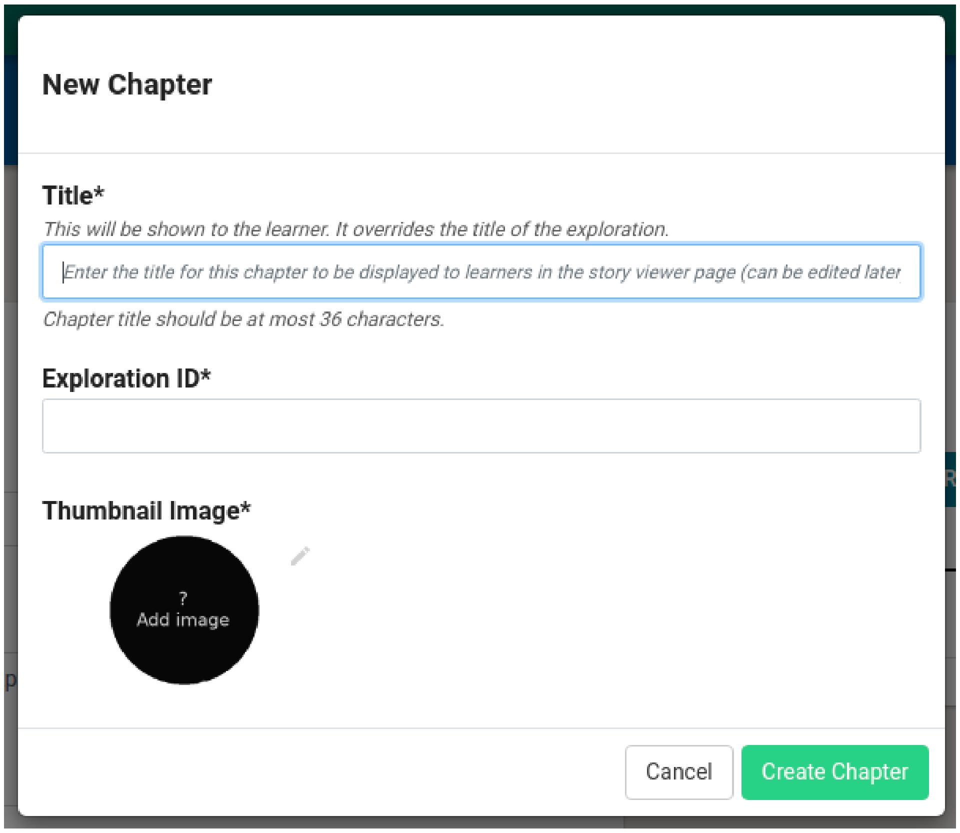
Then add the meta tag and “Page Title Fragment” -- displayed at the navbar and finally add a subtopic. This subtopic requires a skill to be published, to add the subtopic and a skill go to the bottom of the page and add a subtopic and a skill within the subtopic.

After adding the skill assign it to the subtopic


Assign skill to subtopic

[✓ Subtopic]

After publishing the Topic unless there is a story Item, the Topic is not displayed to the end user.
To create a story we have to click the “add story” under ‘’Canonical Stories”


New Chapter

Title*

This will be shown to the learner. It overrides the title of the exploration.

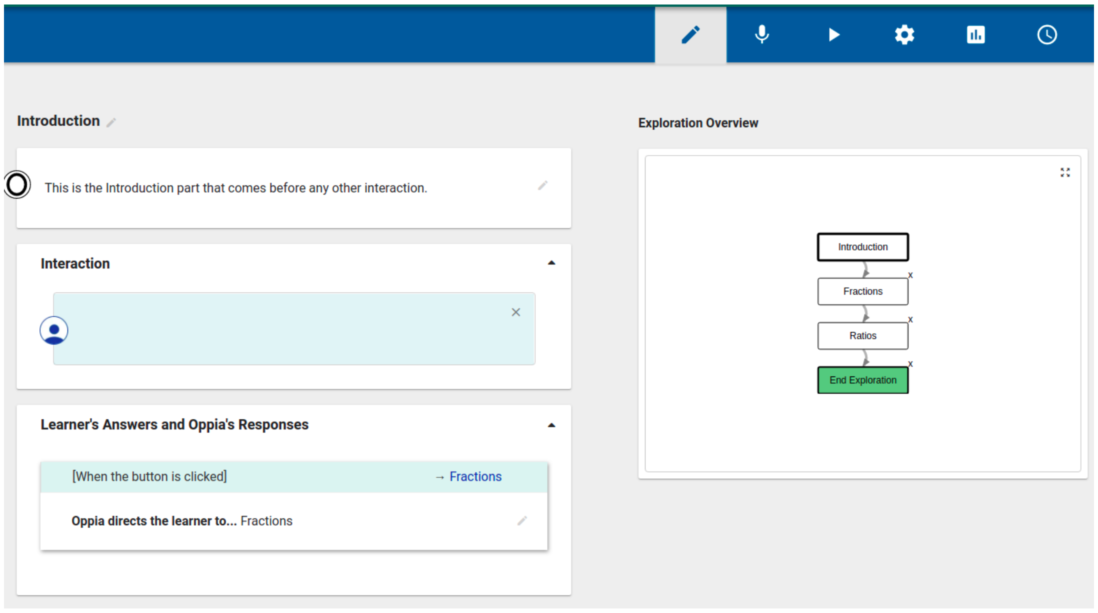
Enter the title for this chapter to be displayed to learners in the story viewer page (can be edited later

Chapter title should be at most 36 characters.

Exploration ID*

Thumbnail Image*

Here the **Exploration ID** has to be the ID of a public pre-published Exploration that can be created using the “Create Exploration” button from /creator-dashboard endpoint.
On Creating Explorations each Page is labeled with the name of the Input Interaction to be tested in Android with a wrong answer feedback.
Example -


Introduction

This is the Introduction part that comes before any other interaction.

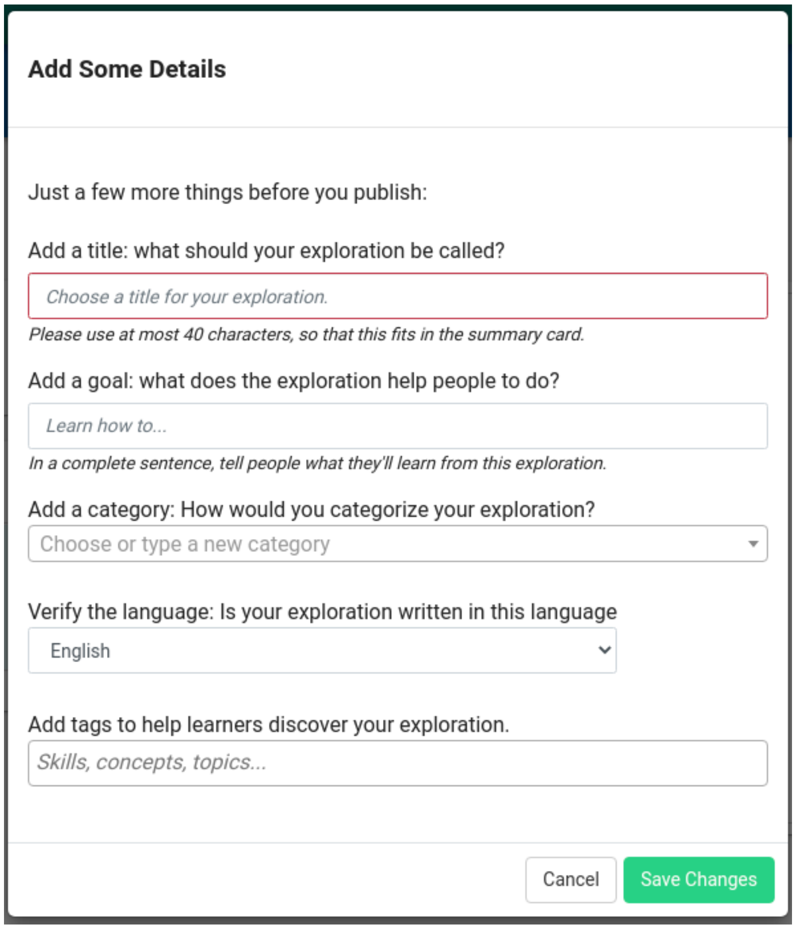
Interaction

Learner's Answers and Oppia's Responses

[When the button is clicked] → Fractions

Oppia directs the learner to... Fractions

After Adding all the pages save the changes using “Save Draft” and add a commit message and publish the exploration with all the necessary Details.


Add Some Details

Just a few more things before you publish:

Add a title: what should your exploration be called?

Choose a title for your exploration.

Please use at most 40 characters, so that this fits in the summary card.

Add a goal: what does the exploration help people to do?

Learn how to...

In a complete sentence, tell people what they'll learn from this exploration.

Add a category: How would you categorize your exploration?

Choose or type a new category

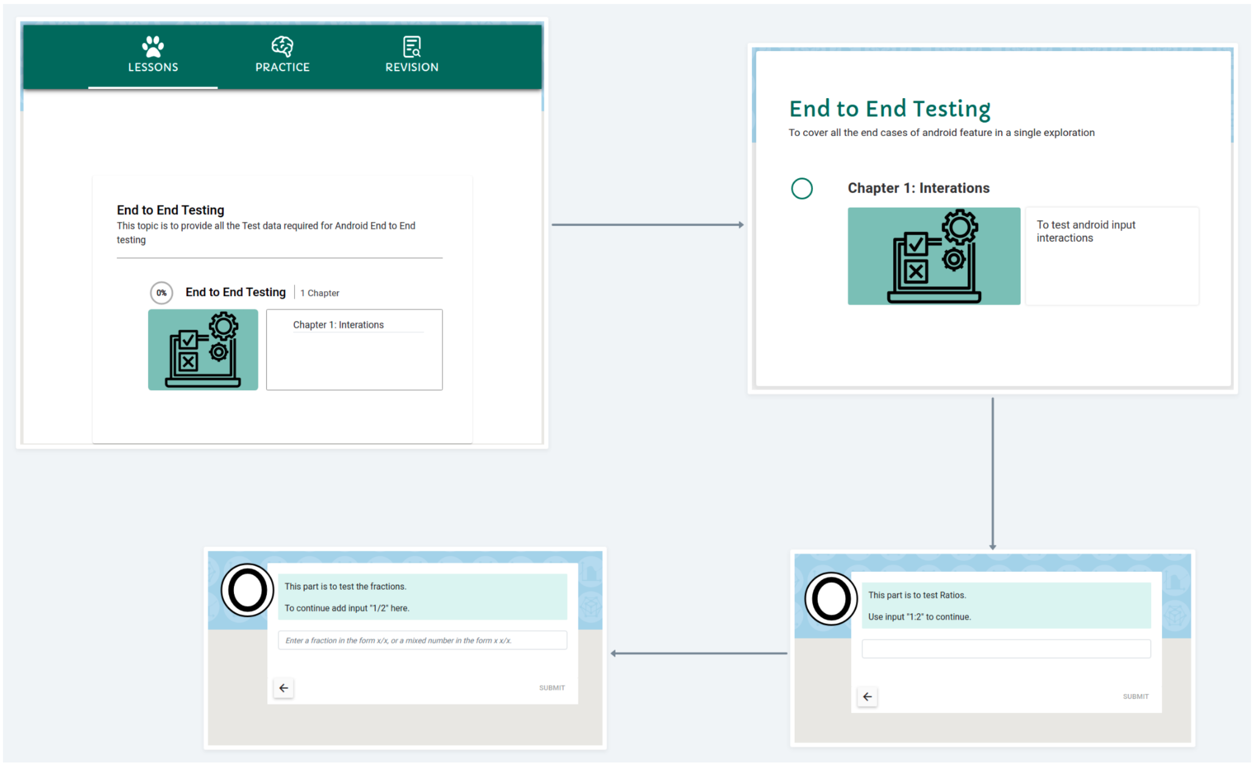
Verify the language: Is your exploration written in this language

English

Add tags to help learners discover your exploration.

Skills, concepts, topics...

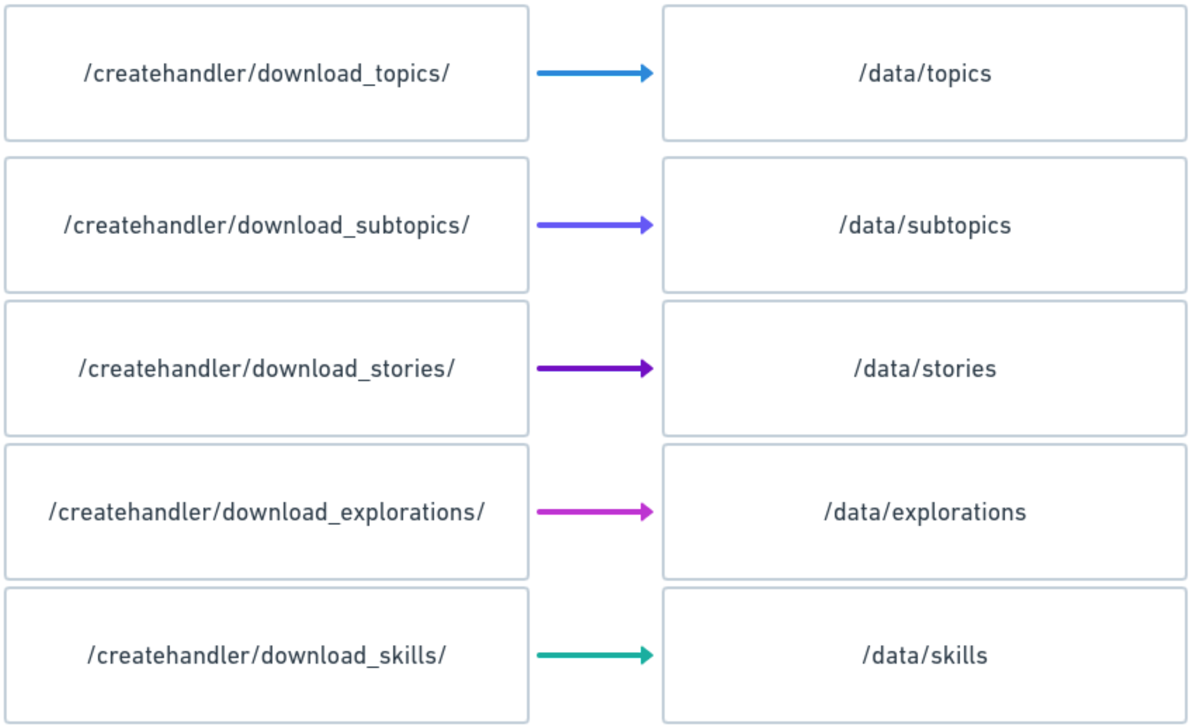
After completing all the above steps on Creating Topics, Explorations, Subtopic, etc., This is how the final Topics flow looks like


LESSONS
PRACTICE
REVISION
End to End Testing
This topic is to provide all the Test data required for Android End to End testing
0%
End to End Testing
1 Chapter
Chapter 1: Interations
End to End Testing
To cover all the end cases of android feature in a single exploration
Chapter 1: Interations
To test android input interactions
This part is to test the fractions.
To continue add input "1/2" here.
Enter a fraction in the form x/x, or a mixed number in the form x x/x.
SUBMIT
This part is to test Ratios.
Use input "1:2" to continue.
SUBMIT

When creating a exploration the flow is as follows

To Download these explorations we have an endpoint /createhandler/download_exploration/<exploration-id> that downloads the Yaml file and image assets in a zip file. Which needs to be extracted and added to local assets in the oppia-web i.e, /data/explorations directory. But here in this project we require the whole topic as a local asset, which can be downloaded from /createhandler/download_topic/<topic-id> and save it to a new directory /data/topics as a representation that it has only test topics and their assets in it.

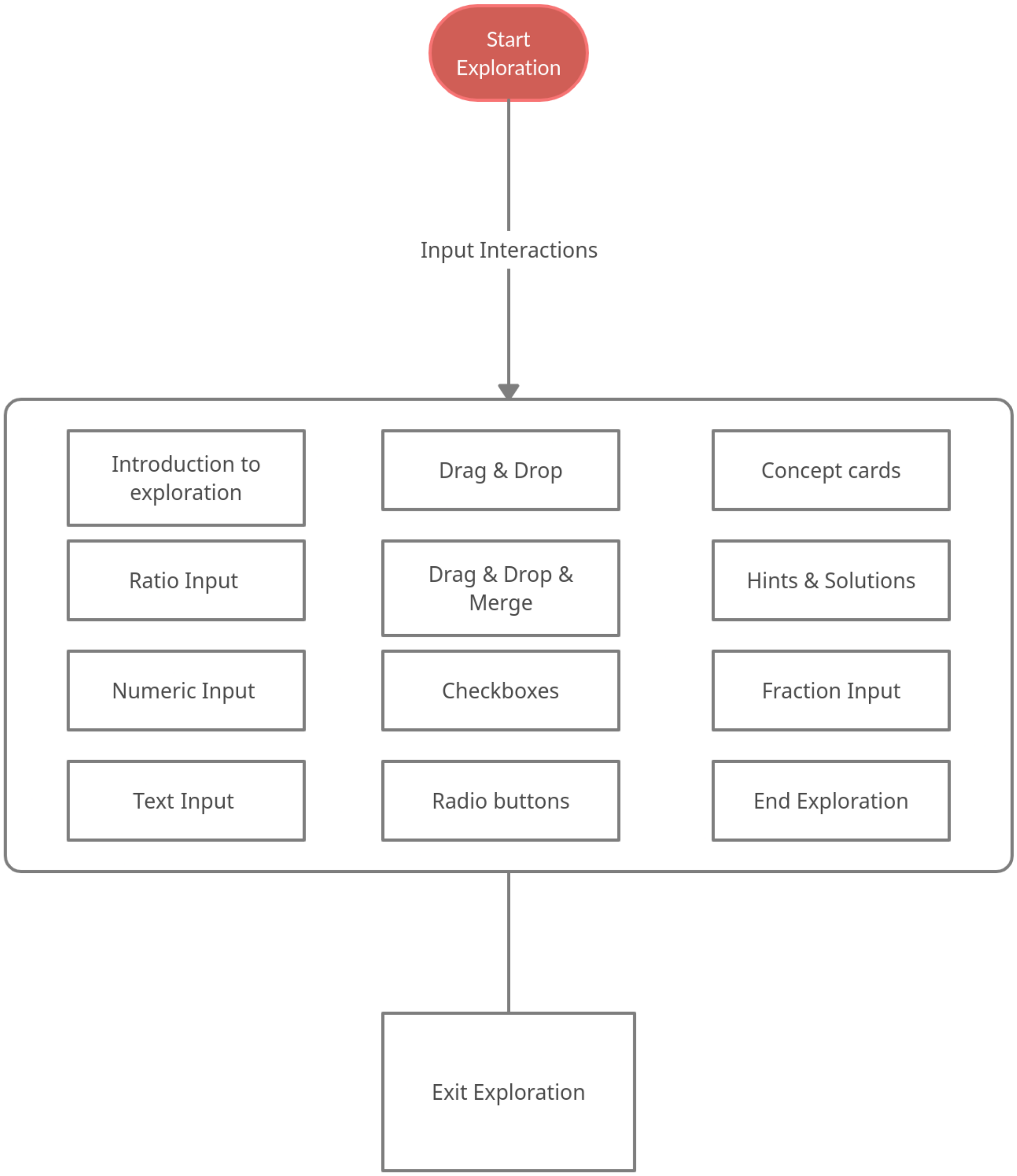
After creating all these Topics, Subtopics, Stories, Explorations, Skills use the above created endpoints to download each of them and save to the respective directory in data directory/module by moving them to the data directory after downloading.


/createhandler/download_topics/
/data/topics
/createhandler/download_subtopics/
/data/subtopics
/createhandler/download_stories/
/data/stories
/createhandler/download_explorations/
/data/explorations
/createhandler/download_skills/
/data/skills

The exploration flow planned to test will be as follows
This will guide on how to create explorations and test them in this project

On opening the exploration,
first displays a “Introduction to the exploration” page describing what the exploration is. After the introduction page, all other pages cover each of the different Input Interactions of android given feedback for wrong answers and hints and solutions for each interaction. There is a concept card for each wrong answer for every page (same concept card for tests purposes).

After finishing all these Input Interactions and close the Exploration.



Initiating the Test Data

Here we need to update the local assets from the /data/topics, /data/explorations, /data/skills, /data/subtopics, /data/stories into the local datastore such that we can fetch all these structures from the datastore through the Android endpoints without changing any APIs in the android project.

This is basically an automated process which saves the topics into the datastore instead of creating new topics manually and publishing every time to run the End to end tests of the 2nd Milestone or to fetch the test data in Android.

To Save local assets to the datastore we need to call the endpoint which fetch the data from the data directory and update them to the datastore

Here we need to be able to save the structure from the local assets into the datastore. So we need to create a function in the core.domain.*_services.py for each structure i.e, save_new_structure_from_yaml_and_assets.

save_new_structure_from_yaml_and_assets

Saves the local data for a structure to the datastore

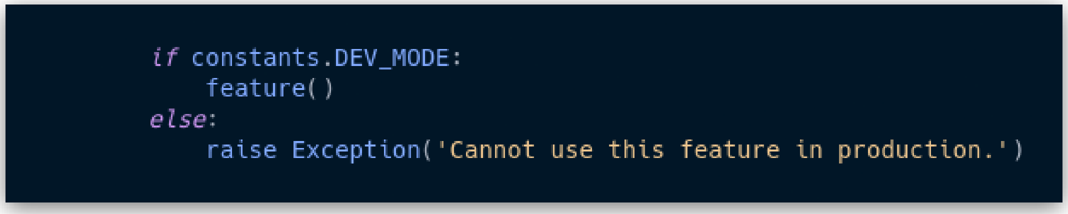
Attribute name	Description
committer_id	The id of the user who made the commit.
yaml_content	The YAML representation of the exploration.
structure_id	The id of the structure.
assets_list	list(list(str)). A list of lists of assets, which contains assets filename and content.
strip_voiceovers	Whether to strip away all audio voice overs from the imported structure.

This function after fetching, parsing, error handling and validations saves the structure into the datastore using the _create_structure function in each of the structure_services.

Now to automate this process we need to trigger this function using an endpoint and hence it requires a new endpoint/controller which will be called using a curl or any other command after starting the server.

Create a new endpoint (i.e, /initiate_test_data) that redirects to InitiateTestDataHandler class in the core.controller.android_initiate_test_explorations.py controller

This InitializeAndroidTestDataHandler fetches all the local test data from the /data/ directory for each structure and calls the save_new_structure_from_yaml_and_assets function to update the datastore. This is only a developer only endpoint and hence we use a flag and an exception such that it wouldn't be used in production.
Example:



```
if constants.DEV_MODE:
    feature()
else:
    raise Exception('Cannot use this feature in production.')
```

Accessing images from the local dev server

Here we need to create two new handler/class AndroidTestImageHandler and AndroidTestThumbnailHandler in the core/controllers/android_e2e_config.py as a mock GCS to fetch the images from the endpoint as a Default GCS prefix to load the images.

This introduces two endpoints in the main.py each for images and thumbnails.

This AndroidTestImagesHandler & AndroidTestThumbnailHandler is flagged using the constants.DEV_MODE to avoid unauthorized access.

This handler fetches images from the local assets (i.e, data directory) and are separated as two endpoints for Images and Thumbnails

For images:
%s/%s/assets/image/%s

For thumbnails:
%s/%s/assets/thumbnail/%s

In the above mentioned urls
The first parameter is the Entity Type/Structure
The Second parameter is the Entity ID/Structure ID
And the last parameter is the url string

Example:
https://storage.googleapis.com/oppiaserver-resources/story/RRVMHsZ5Mobh/assets/thumbnail/img_20200521_034635_46gl425jp5_height_324_width_432.svg

Here
“storage.goolgeapis.com/oppiaserver-resources” is the Image Download Url Template i.e, Default GCS Prefix

“story” is the entity type or structure

“RRVMHsZ5Mobh” is the story ID

img_20200521_034635_46gl425jp5_height_324_width_432.svg is the url string

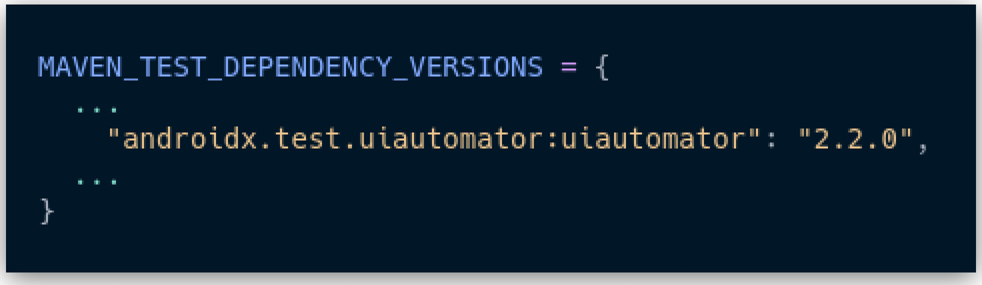
This url in the local server is referred as following
http://localhost:8181/android_test_images/assets/thumbnail/img_20200521_034635_46gl425jp5_height_324_width_432.svg

And the in the app using the proxy server the endpoint is fetched using
http://oppia.org/android_test_images/assets/thumbnail/img_20200521_034635_46gl425jp5_height_324_width_432.svg

And finally add this endpoint to provide Default GCS prefix in the ImageParsingModule to load the images in android using UrlImageParser.

Setting up UI Automator + Bazel

How to start:-
First, we need to add UI Automator dependency to versions.bzl which will be used as a dep for the targets in End-to-End tests.



```
MAVEN_TEST_DEPENDENCY_VERSIONS = {
  ...
    "androidx.test.uiautomator:uiautomator": "2.2.0",
  ...
}
```

Creating a top-level **Instrumentation** directory for the End-to-End tests.

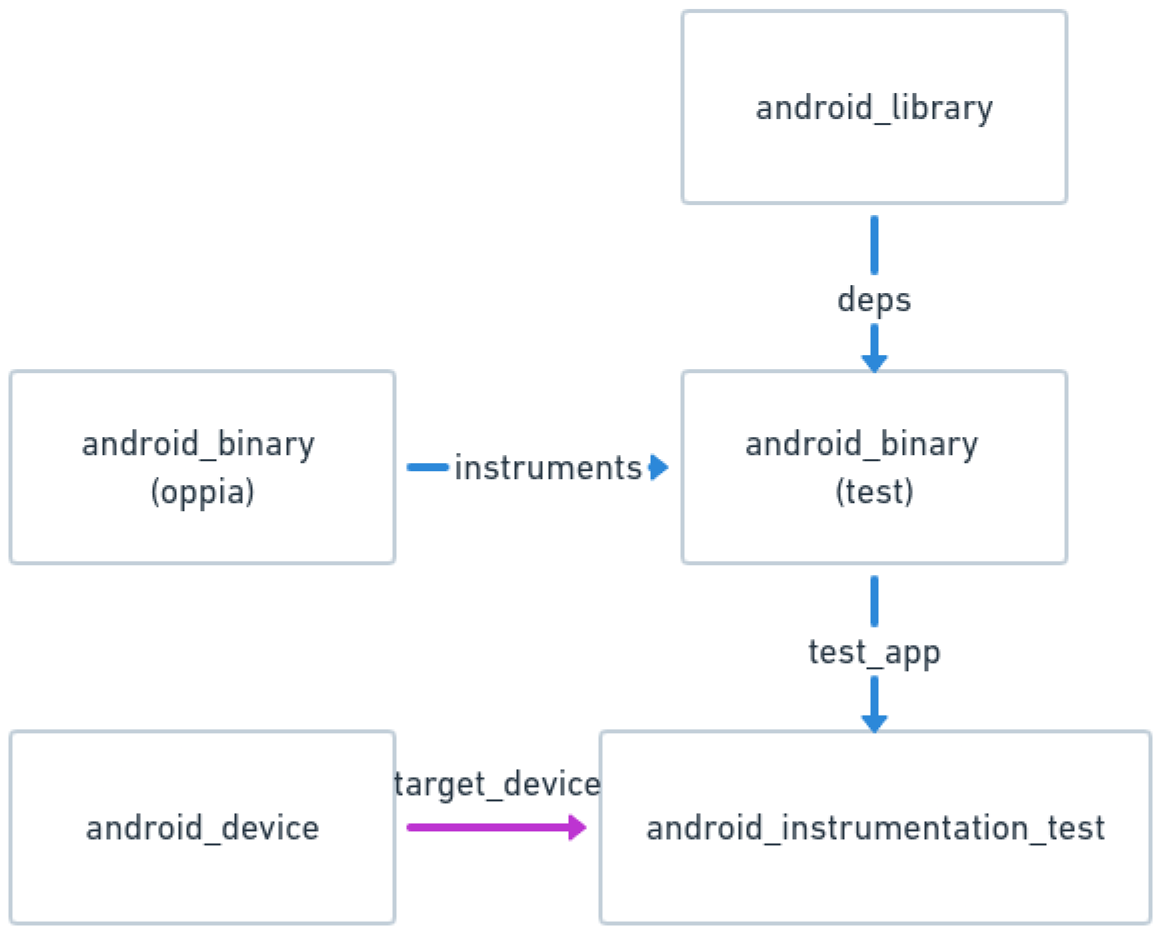
Creating a new file named **tests.bzl** in **instrumentation** similar to *third_party/version.bzl* which contains all the Test suites names to make it easier to run the tests to be given as *name* and *test_app* for *android_instrumentation_test*.

Creating a new file **BUILD.bazel** in **instrumentation** for all the kt_android_libraries within the instrumentation package.

How to setup instrumentation Tests using Bazel

To setup Instrumentation Tests using Bazel we require the following Rules/targets.

- **android_instrumentation_test**
  - name
  - target_device
  - test_app
- **android_binary** [Test]
  - name
  - custom_package
  - instruments
  - manifest
  - Deps
- **kt_android_library**
  - name
  - srcs
  - custom_package
  - manifest
  - resource_files
  - deps



Above flowchart represents how the above android_rules are dependent on each other.

android_instrumentation_test

An *android_instrumentation_test* rule runs Android Instrumentation Tests as the name suggests. It will start an emulator, install the application being tested, the test application, and any other needed applications, and run the tests defined in the test package. The test_app for this rule is a android_binary with test dependencies and sources or test suites to be tested.

Arguments

Attribute name	Usage
name	A unique name or label for the target
target_device	android_device where the tests should run on.

	Note: android_device here is not an emulator or a physical device. There is a different approach for that.
test_app	Label for the android_binary containing the test classes

android_binary

To Produce an Android application package file (.apk) both signed and unsigned.

Arguments

Attribute name	Usage
name	A unique name or label for the target
custom_package	Java/Kotlin package for which java sources will be generated. By default the package is inferred from the directory where the BUILD file containing the rule is. You can specify a different package but this is highly discouraged since it can introduce classpath conflicts with other libraries that will only be detected at runtime.
instruments	If this attribute is set, this android_binary will be treated as a test application for instrumentation tests. An android_instrumentation_test target can then specify this target in its test_app attribute.
manifest	The name of the Android manifest file, normally AndroidManifest.xml. Must be defined if resource_files or assets are defined.
deps	List of libraries i.e, dependencies for the srcs Eg: Targets, artifacts

kt_android_library

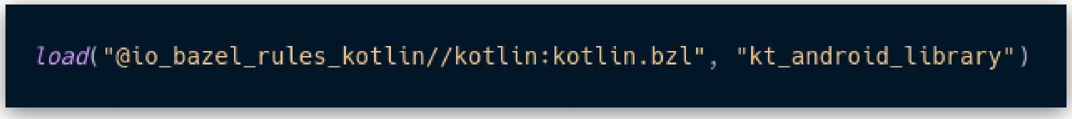
This Rule comes under Rules Kotlin that supports the basic paradigm of Bazel language rules, In kt_android_libary the srcs, deps, plugins are routed to kt_jvm_library.

Simply it is android_library for kotlin, which can compile the kotlin sources into a .jar file. android_library compiles and archives its sources into a .jar file.

Arguments

Attribute name	Usage
name	A unique name or label for the target
srcs	List of ***.kt** files that needs to be processed as a target or create a .jar file
visibility	To alter the visibility of the target beyond the package
deps	List of libraries i.e, dependencies for the srcs Eg: Targets, artifacts

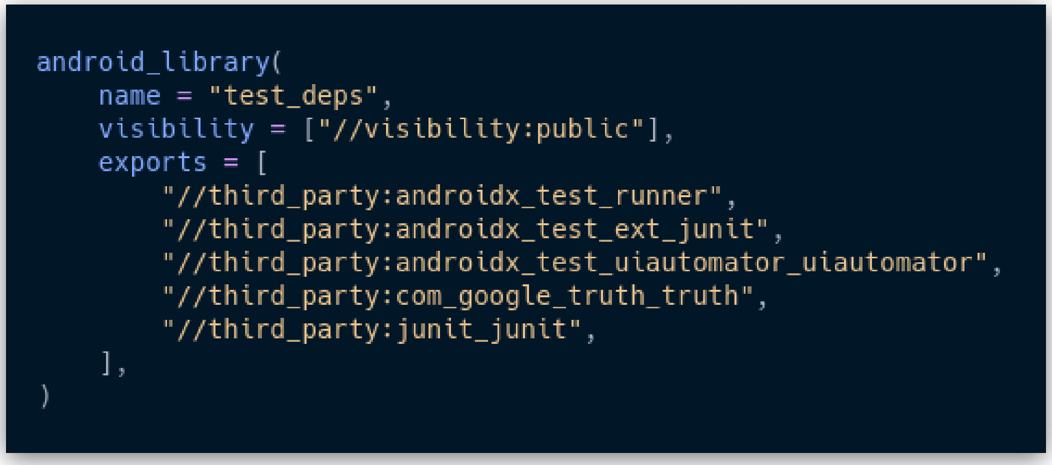
Add the following line in *instrumentation/BUILD.bazel* to support the kt_android_library



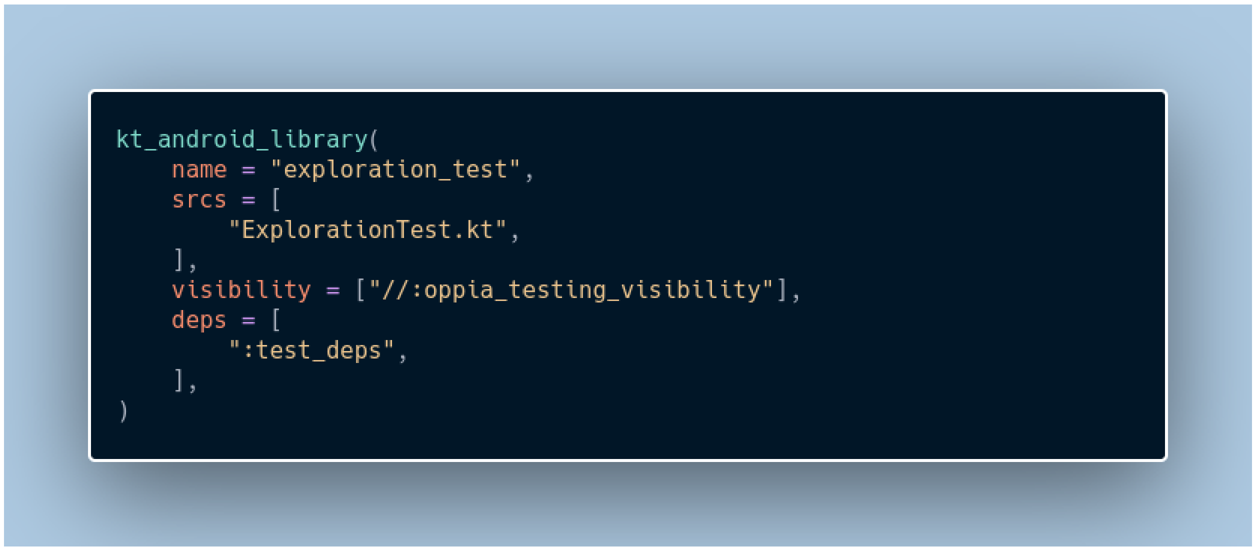
```
load("@io_bazel_rules_kotlin//kotlin:kotlin.bzl", "kt_android_library")
```

Add Test Deps for each test suite or complete instrumentation package
These deps are targets from the third_party packages they are

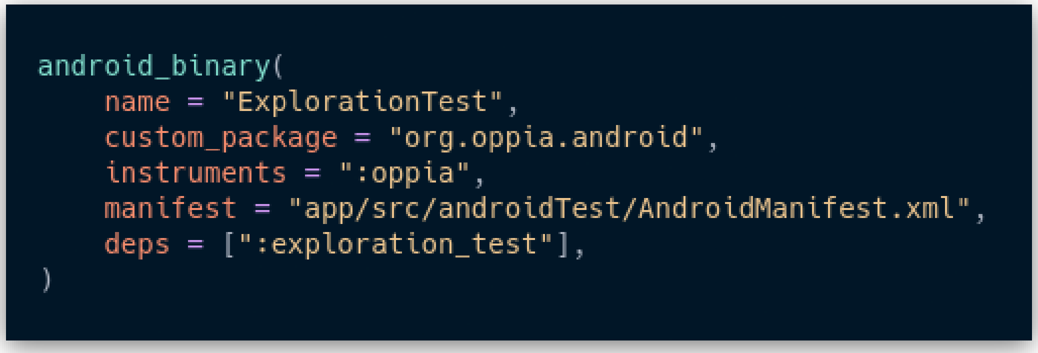
- androidx_test_runner
- androidx_test_ext_junit
- androidx_test_uiautomator_uiautomator
- com_google_truth_truth
- junit_junit


```
android_library(
    name = "test_deps",
    visibility = ["//visibility:public"],
    exports = [
        "//third_party:androidx_test_runner",
        "//third_party:androidx_test_ext_junit",
        "//third_party:androidx_test_uiautomator_uiautomator",
        "//third_party:com_google_truth_truth",
        "//third_party:junit_junit",
    ],
)
```

Let’s assume there is a Test Suite (ExplorationTest) to be tested using bazel, we first need the kt_android_library for that test case.
Eg.

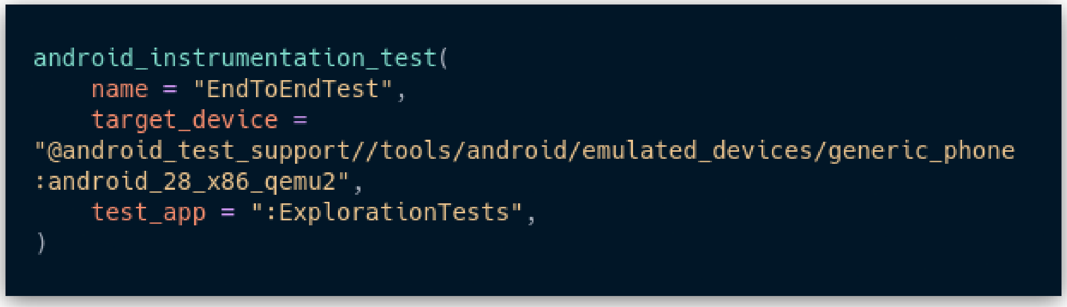

```
kt_android_library(
    name = "exploration_test",
    srcs = [
        "ExplorationTest.kt",
    ],
    visibility = ["//:oppia_testing_visibility"],
    deps = [
        ":test_deps",
    ],
)
```

For the above test suite we need a android test apk, this can be generated using the android_binary
Eg.



```
android_binary(
    name = "ExplorationTest",
    custom_package = "org.oppia.android",
    instruments = ":oppia",
    manifest = "app/src/androidTest/AndroidManifest.xml",
    deps = [":exploration_test"],
)
```

To execute or run the above test apk, we need android_instrumentation_test, which runs the test in a emulator or a local device
Eg.



```
android_instrumentation_test(
    name = "EndToEndTest",
    target_device =
"@android_test_support//tools/android/emulated_devices/generic_phone
:android_28_x86_qemu2",
    test_app = ":ExplorationTests",
)
```

Finally these tests can be run using the command

bazel test EndToEndTest --config=headless (creates a emulator internally but not displayed)

bazel test EndToEndTest --config=gui (creates a emulator internally and displayed)

bazel test EndToEndTest --config=local_device (run on local device or emulator)

The above mentioned configurations are to be added to project's .bazelrc file

1. For config=headless

test:headless --test_arg=--enable_display=false

2. For config=gui

   test:gui --test_env=DISPLAY
   test:gui --test_arg=--enable_display=true

3. For config=local_device

   test:local_device --test_strategy=exclusive
   test:local_device --test_arg=--device_broker_type=LOCAL_ADB_SERVER

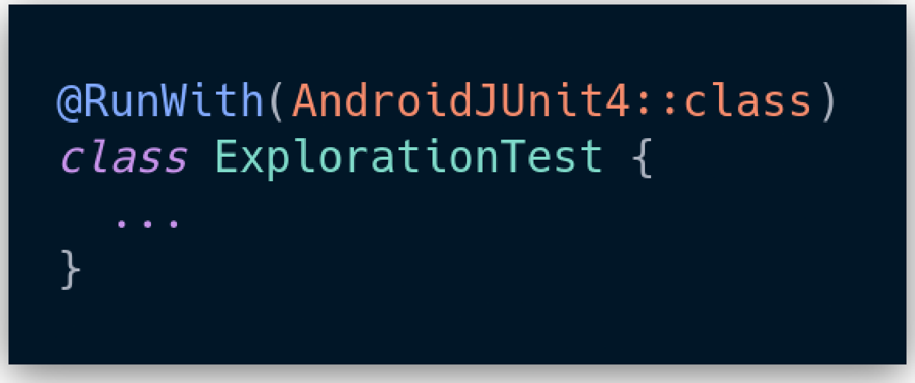
Testing with UI Automator

In order to use the above bazel setup for E2E tests we need few test cases to implement using UI automator. Basically we need to setup Tests using UI Automator.

Create a Test suite ExplorationTest, Instead of the whole End to end Testing lets just test a single and simple test case to verify if the test case is working.

Starting with the
Let the initial test be to check if the "Select your profile" TextView is visible after loading the app.

Create a ProfileTest file/class under the package add the @RunWith annotation to the class.



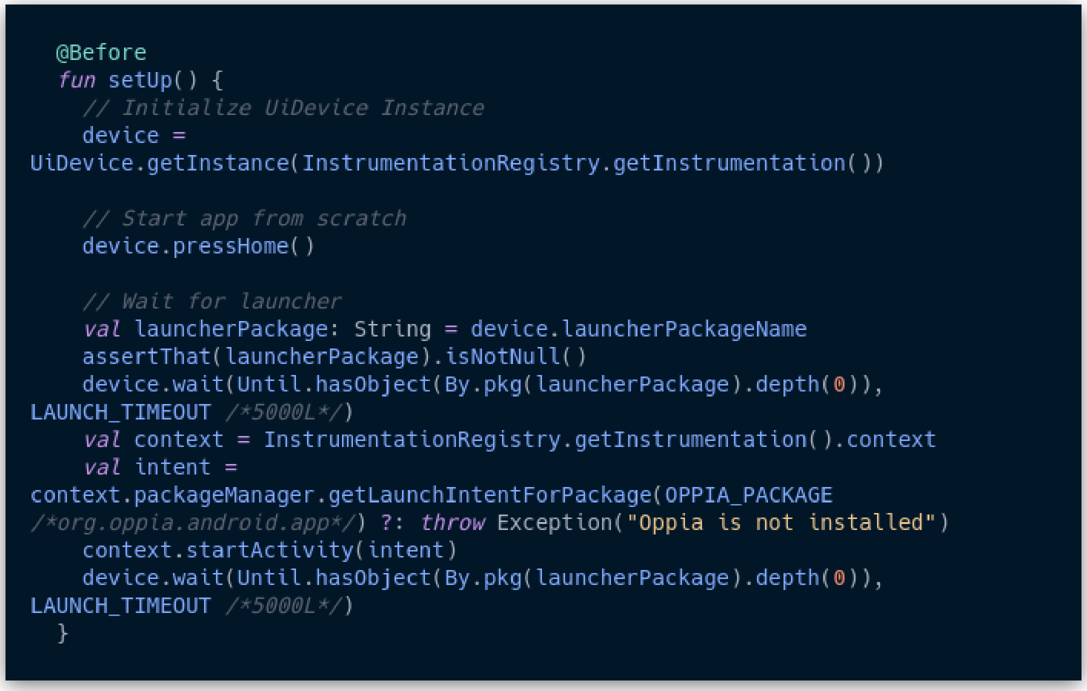
```
@RunWith(AndroidJUnit4::class)
class ExplorationTest {
   ...
}
```

To open the app using UI Automator we need to perform the following

- Get a instance of UIDevice using InstrumentationRegistry
- Start the app from scratch (best for End-to-End testing)
- Get launcher package using UIDevice
- Get Context from InstrumentationRegistry
- Get intent from the Oppia package using context.packagemanager
- Load the intent using context.startActivity

- Wait for the package to load using uiautomator.Until

Following the above steps we get the following **@Before** function which will be executed before each test case.

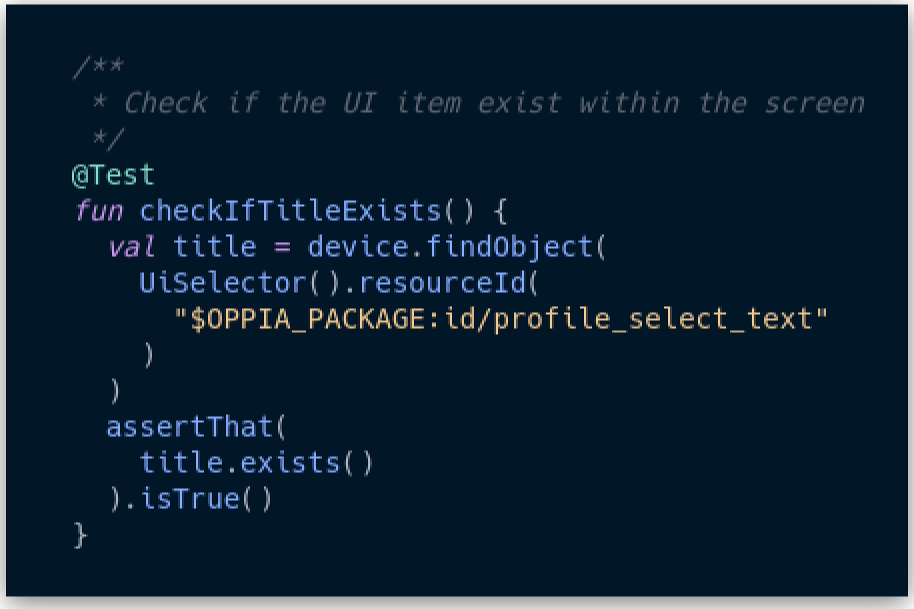


```
  @Before
  fun setUp() {
    // Initialize UiDevice Instance
    device =
UiDevice.getInstance(InstrumentationRegistry.getInstrumentation())

    // Start app from scratch
    device.pressHome()

    // Wait for launcher
    val launcherPackage: String = device.launcherPackageName
    assertThat(launcherPackage).isNotNull()
    device.wait(Until.hasObject(By.pkg(launcherPackage).depth(0)),
LAUNCH_TIMEOUT /*5000L*/)
    val context = InstrumentationRegistry.getInstrumentation().context
    val intent =
context.packageManager.getLaunchIntentForPackage(OPPIA_PACKAGE
/*org.oppia.android.app*/) ?: throw Exception("Oppia is not installed")
    context.startActivity(intent)
    device.wait(Until.hasObject(By.pkg(launcherPackage).depth(0)),
LAUNCH_TIMEOUT /*5000L*/)
  }
```

Adding an initial test case on the basis of starting the End-to-End testing. A test case verifying whether the “Select your profile” exists or not.



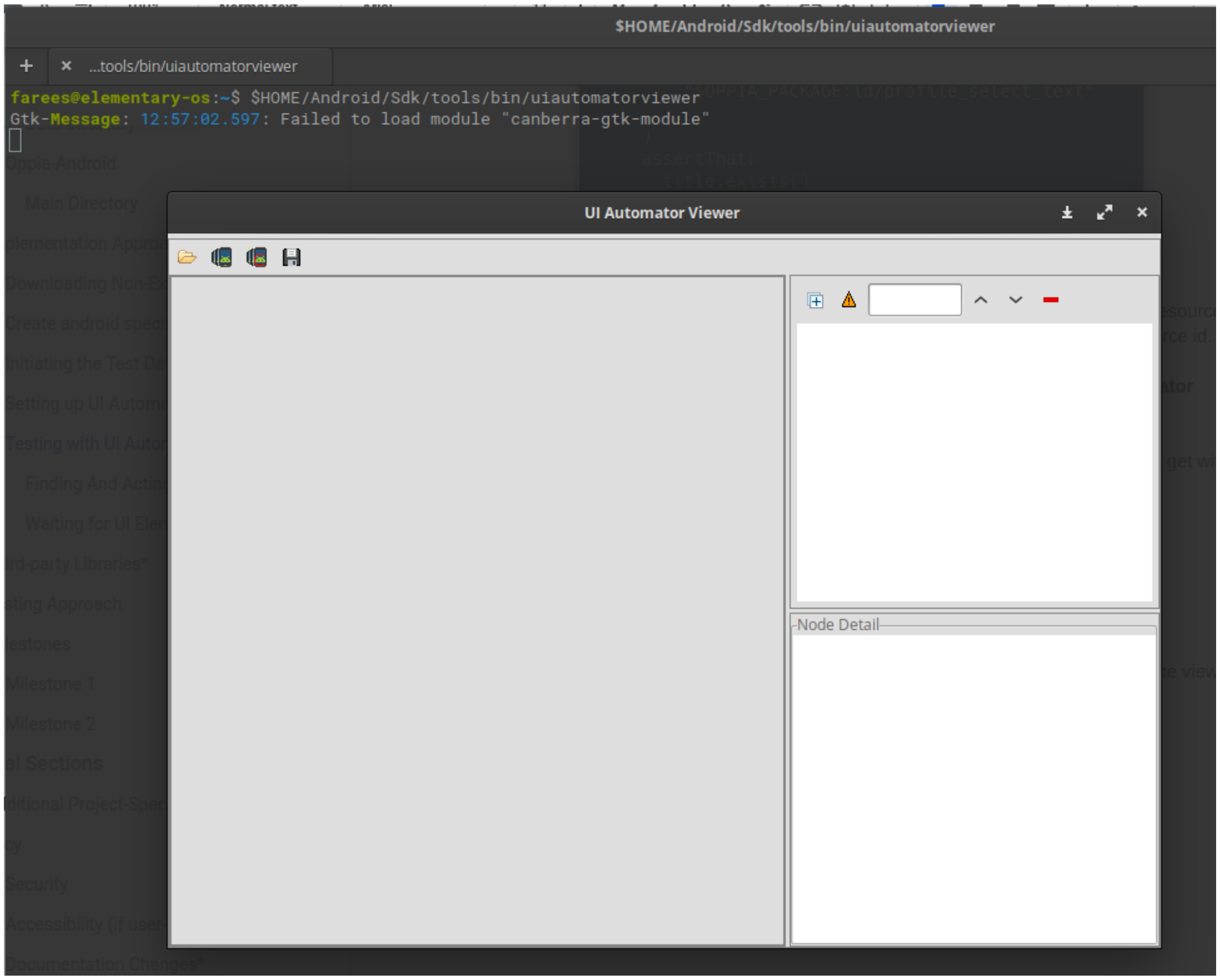
```
/**
 * Check if the UI item exist within the screen
 */
@Test
fun checkIfTitleExists() {
  val title = device.findObject(
    UiSelector().resourceId(
      "$OPPIA_PACKAGE:id/profile_select_text"
    )
  )
  assertThat(
    title.exists()
  ).isTrue()
}
```

Unlike Espresso here we don’t need to `import org.oppia.android.R` Import for the Resource ids instead we can directly use <PACKAGE_NAME>:id/<RESOURCE_ID> for the Resource id. But this makes it difficult as compared to Espresso where we can just use the auto import/suggestion to get the resource id. This issue can be solved through **Ui Automator Viewer**.

UI Automator Viewer is not any third party app or tool. It is a pre-installed Sdk tool we get within the Android Sdk in the path ***/$HOME/Android/Sdk/tools/bin*** i.e, run the command

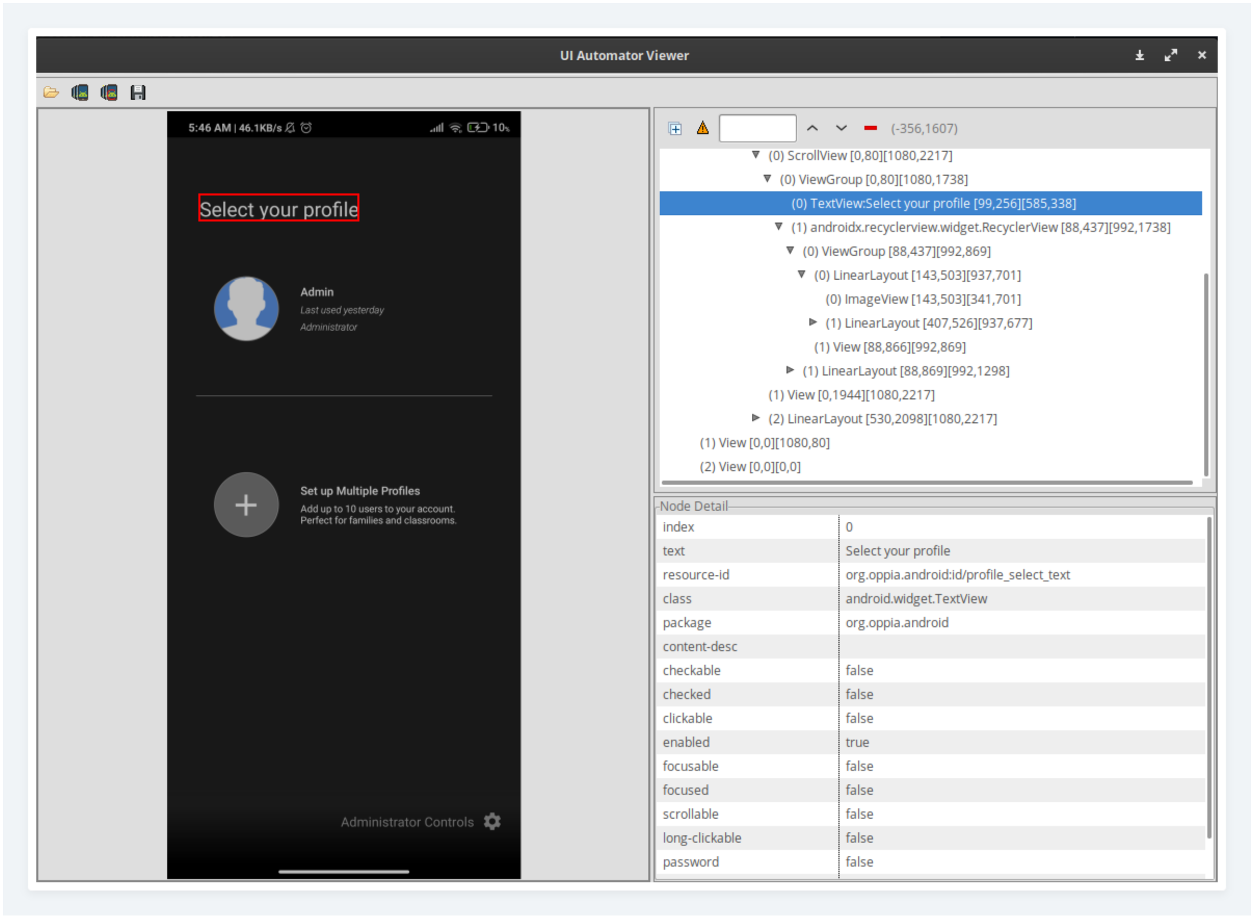
```
$HOME/Android/Sdk/tools/bin/uiautomatorviewer
```

(NOTE: It requires JDK 8 to run.)


$HOME/Android/Sdk/tools/bin/uiautomatorviewer
...tools/bin/uiautomatorviewer
UI Automator Viewer
Node Detail

```
farees@elementary-os:~$ $HOME/Android/Sdk/tools/bin/uiautomatorviewer
Gtk-Message: 12:57:02.597: Failed to load module "canberra-gtk-module"
```

Using Ui Automator Viewer we can just open the current screen and check the id of the view or view group using a single click.



```
▼ (0) ScrollView [0,80][1080,2217]
  ▼ (0) ViewGroup [0,80][1080,1738]
      (0) TextView:Select your profile [99,256][585,338]
    ▼ (1) androidx.recyclerview.widget.RecyclerView [88,437][992,1738]
      ▼ (0) ViewGroup [88,437][992,869]
        ▼ (0) LinearLayout [143,503][937,701]
            (0) ImageView [143,503][341,701]
          ▶ (1) LinearLayout [407,526][937,677]
          (1) View [88,866][992,869]
      ▶ (1) LinearLayout [88,869][992,1298]
  (1) View [0,1944][1080,2217]
▶ (2) LinearLayout [530,2098][1080,2217]
(1) View [0,0][1080,80]
(2) View [0,0][0,0]
```

Node Detail

index	0
text	Select your profile
resource-id	org.oppia.android:id/profile_select_text
class	android.widget.TextView
package	org.oppia.android
content-desc	
checkable	false
checked	false
clickable	false
enabled	true
focusable	false
focused	false
scrollable	false
long-clickable	false
password	false

The above test brings up only one check i.e, to verify if a view is existing or not, but how do we navigate throughout the app similar to View Actions in Espresso.

UI Automator can do all tasks that Espresso can do but not all UI Automator features are supported by Espresso

Analyzing the UI Automator from Espresso perspective as we already have Espresso support in Oppia and we are familiar with Espresso.

- Handling application transitions (navigating to different activity or fragment)
  - Espresso -- Handles windows transitions automatically.
  - UI Automator -- Unlike Espresso we should explicitly use waitings for the application transitions.
- Locating UI elements (Views or View groups)
  - Espresso -- Core Espresso onView() and onData() methods together with view matchers or data matchers are used to locate the UI elements
  - UI Automator -- Similar to Espresso, UI Autmator has its own UIDeivce class, which contains methods like hasObject(BySelector), findObject(BySelector) that are used to search for an element or check its presence in the app.
- Waitings (to wait for a specific tasks to complete)

    - Espresso -- The Test coroutine Dispatcher or IdlingResource and the third-party ConditionWatcher can be used as waiting mechanisms.
    - UI Automator -- Unlike Espresso, UI Automator has its own function wait() and performActionAndWait() within the UIDevice class.
- UI/View Actions. (Interacting with views)
    - Espresso -- Core Espresso ViewActions do all the view actions for the tests.
    - UI Automator -- Similar to Espresso, UI Automator has a UIObject2 Class that allows all the UI Actions.
- Device Controls (Other controls such as volume, home button...)
    - Espresso -- Not supported in Espresso
    - UI Automator -- Provides a long list of device control methods starting from home button to orientation control.

The above differences lists a lot of unknown functions/classes related to UI Automator framework.

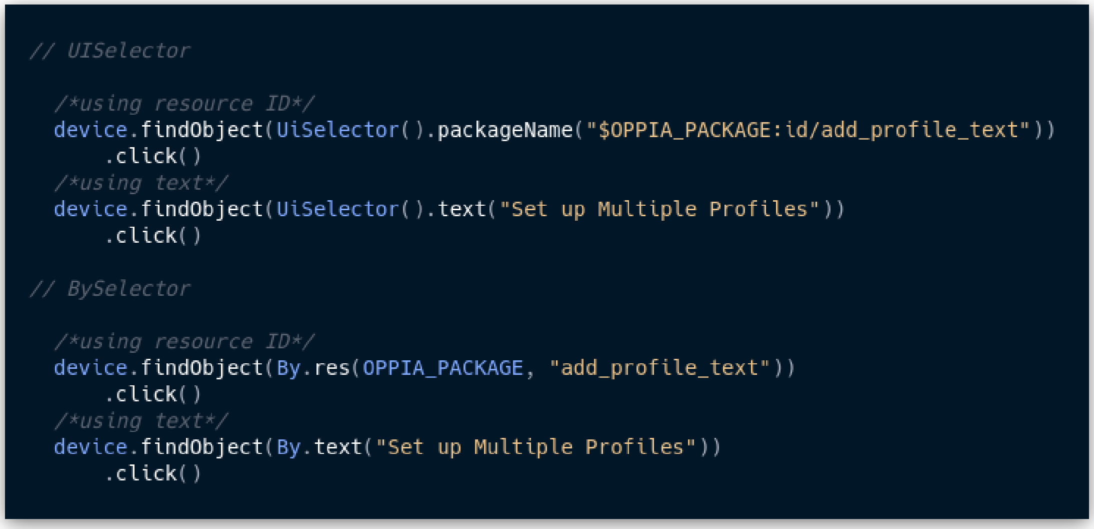
Among the above features, we need to understand the following two important features to be able to write any test case using UI Automator.
- Finding and Acting on UI Elements
- Waiting for UI Elements

Finding And Acting on UI Elements

This is the main UI Automator functionality is to locate UI Elements and performing actions on them. To do this we need findObject() and findObjects() methods. The findObject() method takes instances of the following classes as parameters that specify criteria for matching UI elements in the hierarchy:
- UiSelector()
- BySelector()

Both UISelector() and BySelector() are completely similar to one another. But BySelector() is more readable compared to UISelector.

Usage:



```
// UISelector

  /*using resource ID*/
  device.findObject(UiSelector().packageName("$OPPIA_PACKAGE:id/add_profile_text"))
      .click()
  /*using text*/
  device.findObject(UiSelector().text("Set up Multiple Profiles"))
      .click()

// BySelector

  /*using resource ID*/
  device.findObject(By.res(OPPIA_PACKAGE, "add_profile_text"))
      .click()
  /*using text*/
  device.findObject(By.text("Set up Multiple Profiles"))
      .click()
```

Waiting for UI Elements

UI Automator interacts with multiple applications unlike Epsreso so we don't have access to the Applications transitions or animations -- it is important to have proper waiting mechanisms that will allow us to write more reliable test code.

UI Automator waitings are presented by the following types:

- EventCondition -- A condition that depends on a event to be occurred.
- SearchCondition -- A condition that is satisfied by searching for UI elements.
- UiObject2Condition -- A condition that is satisfied when a UIObject2 is in a Exptected state.

EventCondition is a condition which depends on an event or series of events having occurred. It is a parameter to *clickAndWait(condition: EventCondition<R>, timeout: long)* function of the UIOjbect2 class. The following are the event conditions.

- newWindow() -- Returns a condition that depends on a new window having appeared.
- scrollFinished() -- Returns a condition that depends on a scroll having reached the end in the given direction

SearchCondition is responsible for locating elements in the layout. The mosly used searchConditions are

- gone() -- Condition is satisfied when the UIObject can no longer be found
- hasObject() -- Condition is satisfied when at least one element matching the selector can be found
- findObject() -- Condition is satisfied when at least one element matching the selector can be found and returns the first found element

UIObject2Condition is basically a Static check i.e, it doesn't involve any interaction with the views. They are similar to Espresso's ViewMatchers. The most used ones are

- checkable() -- Returns a condition if the UIObject2 is in a checkable state.
- checked() -- Returns a condition if the UIOject2 is in a checked state.

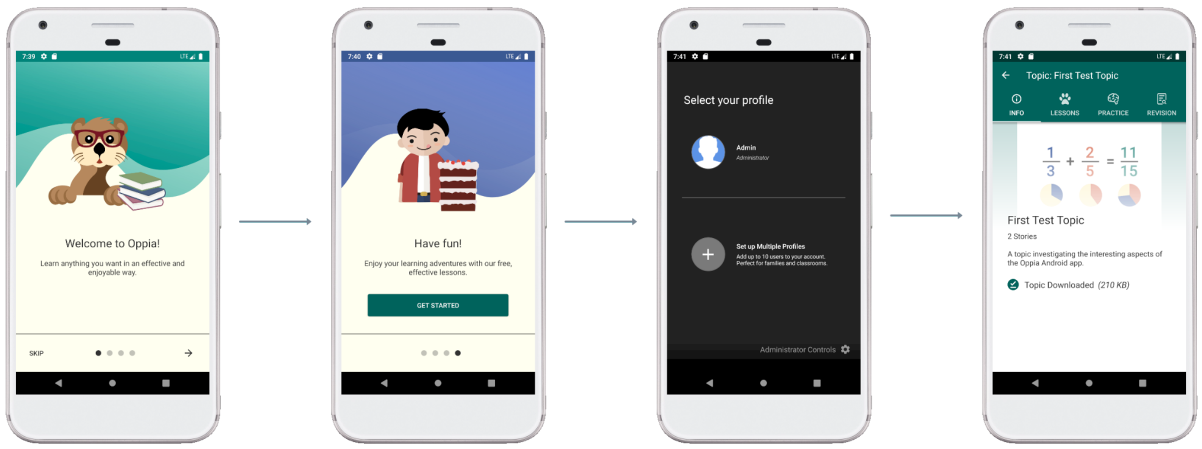
- enabled() -- Returns a condition if the UIObject2 is in an enabled state.
- focused() -- Returns a condition if the UIOject2 is in a focused state.
- selected() -- Returns a condition if the UIObject2 is in the selected state.
- textMatches() -- Returns a condition if the object’s text value matches with the given text.

End-to-End test flows

As this project is basically to test Downloading and playing through an exploration. This section provides a high level representation of how the testing flow takes place.

The test cases are written such that the @Before annotated function will open a topic so that the tests only include the Exploration related testing (This will also favor the name of the test suite -- ExplorationTests)

@Before
Opens the app from scratch → Selecting a user → choosing a topic


Welcome to Oppia!
Learn anything you want in an effective and enjoyable way.
SKIP
Have fun!
Enjoy your learning adventures with our free, effective lessons.
GET STARTED
Select your profile
Admin
Administrator
Set up Multiple Profiles
Add up to 10 users to your account. Perfect for families and classrooms.
Administrator Controls
Topic: First Test Topic
INFO
LESSONS
PRACTICE
REVISION
First Test Topic
2 Stories
A topic investigating the interesting aspects of the Oppia Android app.
Topic Downloaded (210 KB)

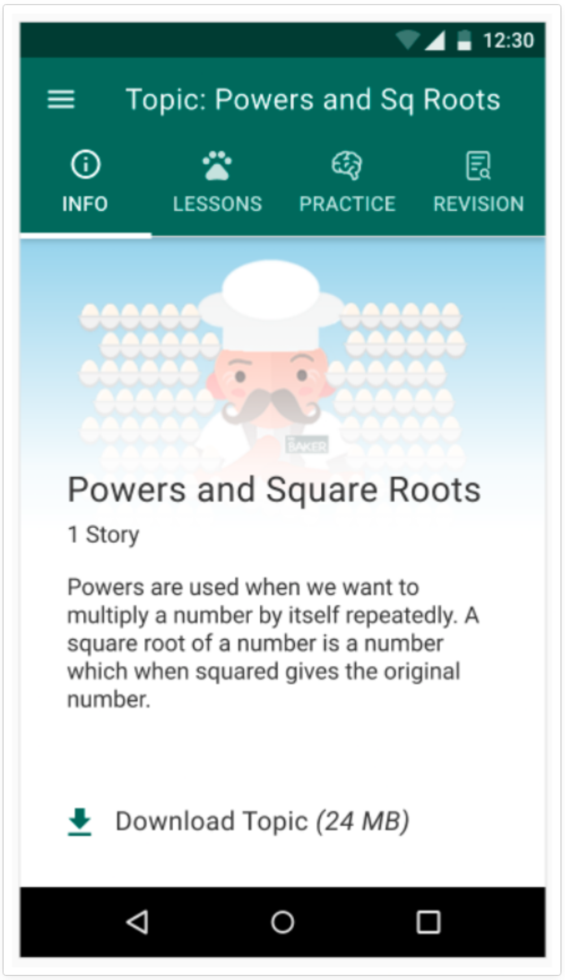
This images is only when the app is open first time further it will be only the last to images. And other will be ignored by using UI Automator’s UIObject2Conditonal waitings.

Downloading Topics

This section includes
- Downloading a Topic
- Pause Download
- Cancel Download
- Delete Downloaded Topic
- Validate Downloading with wifi
- Validate Downloading with Mobile Data
- Validate Downloading without internet connection

- Validate Progress while downloading

Adding screenshots from the mocks as this is not yet implemented

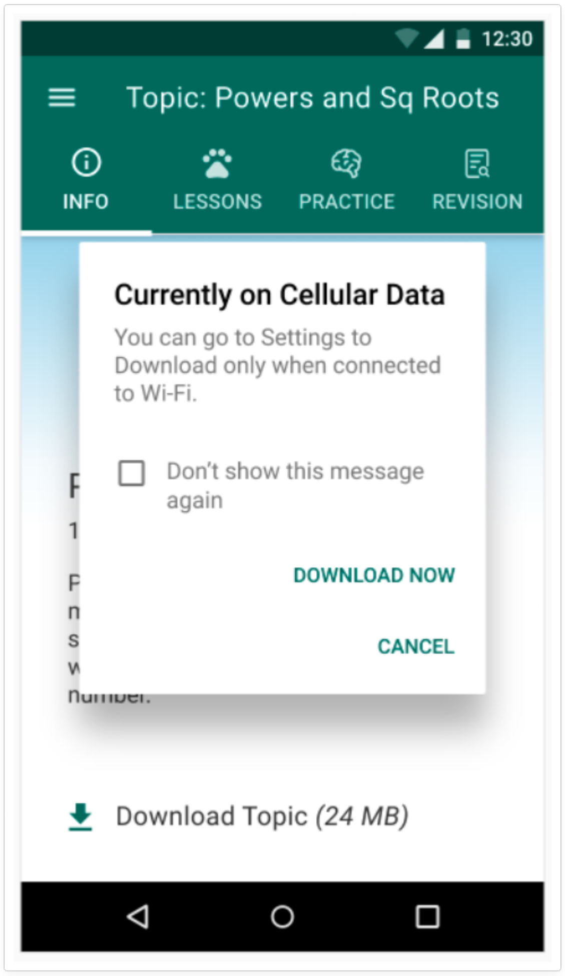


Powers and Square Roots

1 Story

Powers are used when we want to multiply a number by itself repeatedly. A square root of a number is a number which when squared gives the original number.

Download Topic *(24 MB)*



Currently on Cellular Data

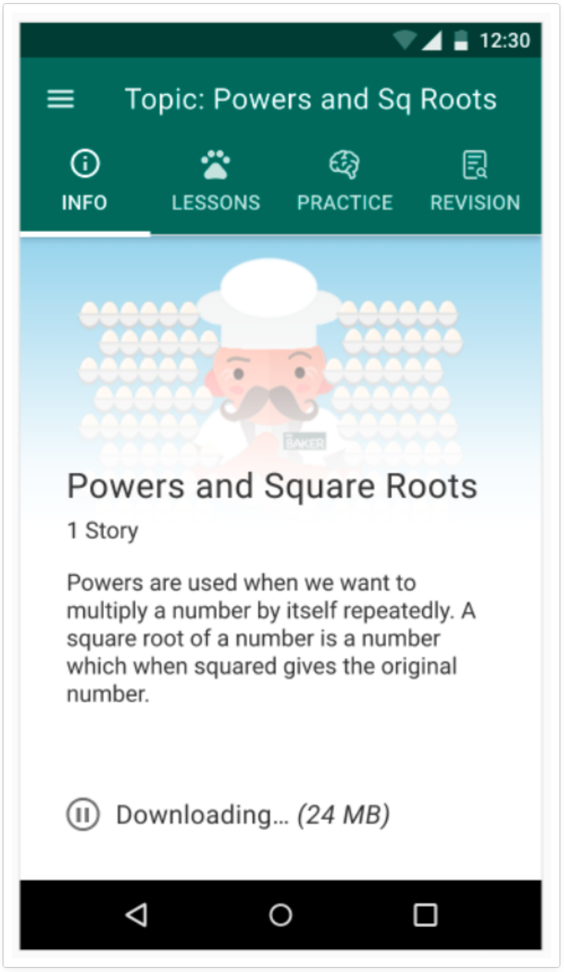
You can go to Settings to Download only when connected to Wi-Fi.

☐ Don't show this message again

DOWNLOAD NOW

CANCEL

Download Topic *(24 MB)*


12:30

Topic: Powers and Sq Roots

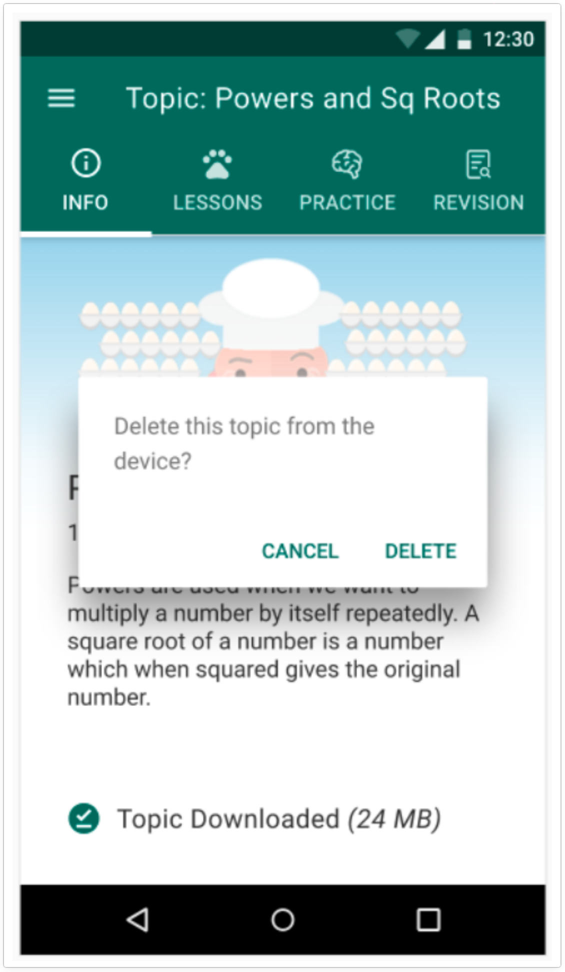
INFO LESSONS PRACTICE REVISION

Powers and Square Roots

1 Story

Powers are used when we want to multiply a number by itself repeatedly. A square root of a number is a number which when squared gives the original number.

Downloading... *(24 MB)*


12:30

Topic: Powers and Sq Roots

INFO LESSONS PRACTICE REVISION

Delete this topic from the device?

CANCEL DELETE

multiply a number by itself repeatedly. A square root of a number is a number which when squared gives the original number.

Topic Downloaded *(24 MB)*

Playing through a exploration

Note: This can be tested after downloading a Topic. Not mentioning the user flow for all these, as it would take a huge amount of test cases.

This section describes what exploration components/Input Interactions in android to be tested

- Fraction Input
- Ratio Input
- Numeric Input
- Text Input
- Drag and drop
- Drag and drop with merge
- Checkbox selection Input
- Radio button selection input
- Concept cards
- Hints and solutions

- Confetti

Third-party Libraries*

This project introduces UI Automator, a Third-party library used for End-to-End (and Instrumentation) tests. UI Automator is compatible with Apache 2.0 license.

UI Automator maven repository

Testing Approach

As the android part of this project is especially based Testing we don't need to include the End to End tests.

But for the web part it is necessary to add unit tests for the controllers and domain module i.e, for the following files.

- core/controllers/editor.py
- core/controllers/acl_decorators.py
- core/controllers/android_e2e_config.py
- core/domain/topic_services.py
- core/domain/topic_fetchers.py
- core/domain/story_services.py
- core/domain/skill_services.py

And the test cases are added under.

- core/controllers/editor_test.py
- core/controllers/acl_decorators_test.py
- core/controllers/android_e2e_config_test.py
- core/domain/topic_services_test.py
- core/domain/topic_fetchers_test.py
- core/domain/story_services_test.py
- core/domain/skill_services_test.py

Milestone 1

Key Objective: Introduce developer-only functionality in the Oppia backend to prepopulate test topics, stories, chapters, explorations, revision cards, skills, and questions.

To Introduce downloadable setup for non-Exploration structures, and update the local datastore using the local assets in order to fetch the data from Oppia-android as an automated process. To Introduce a Mock GCS endpoint used to access test images from the data directory.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
1.1	Introduce Download support for Topics, Subtopics, Stories and skills (new endpoints to download these structures)	N/A	9th June, 2021	13th June, 2021
1.2	Add sample topic, subtopics, stories, skills and explorations to be tested in android	1.1	15th June, 2021	19th June, 2021
1.3	Initiate test data from local assets	1.2	21th June, 2021	24th June, 2021
1.4	Send local test images as a HTTP response	1.2	26th June, 2021	30th June, 2021
1.5	Add the mock GCS endpoint to ImageParsingModule	1.4	1st June, 2021	5th July, 2021

Milestone 2

Key Objective: Set up infrastructure for end-to-end testing using UiAutomator & Bazel. Write end-to-end tests for downloading and playing through one exploration.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
2.1	Setup UI Automator + Bazel	N/A	20rd July, 2021	25th July, 2021
2.2	End-to-End test for downloading a topic	2.1	27th July, 2021	1st August, 2021
2.3	End-to-End test to play through a exploration	2.1	4th August, 2021	9th August, 2021

Optional Sections

Additional Project-Specific Considerations

Privacy

This feature does not collect any data from the user; the test data is fetched from the local assets that were added in the data directory. And hence this project does not deal with any privacy issues for Oppia users

Security

This feature includes adding an endpoint that fetches images and yaml strings from the data directory i.e, initiating the tests data which might break the website if used in production. For security measures the endpoint is surrounded by a flag that checks the user's authorization and throws a exception in case of unauthorized access

Accessibility (if user-facing)

This project doesn't involve only changes in the backend of oppia-web and introduces End-to-End tests in oppia-android, and hence there are no UI Changes involved.

Documentation Changes

The following parts of this project requires Documentation changes.

- How to download test data from the data store.
- How to update test data from local assets in the data directory to the local datastore.
- How to run End-to-End tests using an UI automator.
- How to start UI Automator Viewer.
- How to write End-to-End tests.

Ethics

End-to-End tests are necessary before publishing or adding new features, this is considered a best practice to test End-to-End tests before every release as it helps to cover all the user end cases by replicating user behaviors in an automated and deterministic way allows us to guard against regressions which in turn ensures users receive a consistent and correctly working app for each release..

Future Work

After GSoC I'd love to work on the following things

1. Adding CI checks for UI Automator.

This project doesn’t involve local End to End tests for UI Automator. But further we need to add these to CI check such that after each PR or each change in the codebase these End-to-End tests are run and in case the app breaks or doesn’t give expected outcome, these tests fail pointing to the code changes that triggered the failure.

2. Adding Bazel support to run Android Instrumentation tests for existing Espresso tests.

Currently the Espresso tests are only supported by Gradle, but after bazel migration these Espresso tests need to run on bazel.

Additional Info Required

Technical Design

Q1. The image serving solution seems incomplete: there is no explanation given for how images will be retrieved. Please add more details on how the implementation of "Accessing images from the local dev server" would work.

A1. Please write your answer here.

To retrieve local images from the web this project introduces two handler/class AndroidTestImageHandler and AndroidTestThumbnailHandler in the core/controllers/android_e2e_config.py that act as a fake Google Cloud Storage to the android.

Referring from the android side (ImageParsingModule.kt) two endpoints are created for these two handlers.

Images → android_test_images/%s/%s/assets/image/%s
Thumbnails → android_test_images/%s/%s/assets/thumbnail/%s

Each endpoint gives three parameters

- Structure type
- Structure ID
- File Name

Which are used to navigate to the images file from the test data in the data directory. These images are then expected to be displayed similar to GCS such that this doesn't involve any changes in UrlImageParser.kt in Oppia-android.

So after locating the required image the image path is passed to a new function (render_image) in core.controllers.base.py which loads the images for the endpoint.

In core.controllers.base.py a new function is created i.e, render_image() in which the image file path is passed and it opens the image as a HTML or send the image as a HTTP response.

```
def render_image(self, filepath):
  with open(filepath, "rb") as imagefile:
      image = imagefile.read()
  self.response.content_type = b'image/png'
  self.response.write(image)
```

Using this function the endpoint returns an image response that can be directly retrieved using Glide android image loading library to display the local test images.

Q2. Please explain how image handling will be replaced in the Android app. (Note that it needs to be done in a way that doesn't change production app behavior.)

A2. Please write your answer here.

Test images can be fetched using the above endpoint. To do so we need to change the

Default GCS prefix and Default GCS resource to and use the local server endpoints to access the test images from the data directory in oppia-web. This process of changing the production GCS prefix and resource can't be automated as these are used from modules and needs Hilt migration to do so.

Here in production the GCS prefix and GCS resource are https://storage.googleapis.com and oppiaserver-resources.

This constants are changes to "http://oppia.org" (proxy server for localhost:8181) as GCS prefix in the ImageParsingModule.kt
And android_test_images (new endpoint for images and thumbnails) as GCS resource in the GcsResourceModule.kt

As this has to be done manually this can either be added as a comment or mentioned in the wiki to run the End-to-End tests.

Q3. Please explain which initial test cases will be implemented, along with a code snippet or pseudocode to explain the flow.

A3. Please write your answer here.

The initial test cases for End-to-End testing will be to “Navigating to a topic”, As this project involves testing Downloading a Topic and Playing through a Exploration.

The following code snippets describe which resource ids and class names are used based on the each view as displayed using UiAutomatorViewer and a basic understanding of the test flow through the topics and explorations. Using Prototype Exploration as Example

To Navigate to a Topic.

```
device.findObject(UiSelector().packageName("org.oppia.android:id/profile_ch
ooser_item")).click()
device.findObject(UiSelector().packageName("org.oppia.android:id/topic_thum
bnail_image_view")).click() // Selects the first found topic
```

Downloading a Topic

```
val                               download_status                              =
device.findObject(UiSelector().packageName(”org.oppia.android:id/download_story
_count_text_view”))
assertThat(
    Download_status.text, “Download Topic”
)
download_status.click()
device.findObject(UiSelector().packageName(”android:id/button1”)).click
//Download Now button in dialog
```

Deleting a downloaded Topic.

```
val                               download_status                              =
device.findObject(UiSelector().packageName("org.oppia.android:id/download_story
_count_text_view"))
assertThat(
    Download_status.text, "Topic Downloaded"
)
download_status.click()
```

```
device.findObject(UiSelector().packageName("android:id/button1")).click
//confirm delete topic
```

Opening an Exploration and playing through it.

```
device.swipe(1033,1346,531,1346,20) // swipe to next page or Lessons
device.findObject(UiSelector().className("android.widget.LinearLayout").getInst
ance(0)).click() // Revealing Explorations in first story
device.findObject(UiSelector().packageName("org.oppia.android:id/chapter_contai
ner").getInstance(0)).click() // Opening the first Exploration.
device.findObject(UiSelector().packageName("org.oppia.android:id/continue_butto
n")).click() // continue button
```

Fraction Input Interaction.

```
val fraction_input =
device.findObject(UiSelector().resourceId("org.oppia.android:id/fraction_input
_interaction_view"))

fraction_input.setText("1/2")
```

Radio Button Interaction.

```
val correct _option =
device.findObject(UiSelector().resourceId("org.oppia.android:id/radio_contain
er").getInstance(2))

correct_option.click()
```

Checkbox Interaction

```
val                                correct_option                               =
device.findObject(UiSelector().resourceId("org.oppia.android:id/checkbox_contai
ner").getInstance(0))

correct_option.click()

device.findObject(UiSelector().resourceId("org.oppia.android:id/submit_answer_b
utton")).click() //submit answer
```

Numeric Input Interaction

```
val                                numeric_input                               =
device.findObject(UiSelector().resourceId("org.oppia.android:id/numeric_input_
interaction_view"))

numeric_input.setText("121")
```

Ratio Input Interaction

```
val                                ratio_input                                =
device.findObject(UiSelector().resourceId("org.oppia.android:id/ratio_input_i
nteraction_view"))

ratio_input.setText("4:5")
```

Text Input Interaction

```
val                                text_input                                 =
device.findObject(UiSelector().resourceId("org.oppia.android:id/text_input_in
teraction_view"))

ratio_input.setText("finnish")
```

Drag & Drop Interaction

```
val init_position =
device.findObject(UiSelector().resourceId("org.oppia.android:id/drag_drop_item_
recyclerview").getInstance(0))

val des_position =
device.findObject(UiSelector().resourceId("org.oppia.android:id/drag_drop_item_
recyclerview").getInstance(3))

init_positon.dragTo(des_position)

device.findObject(UiSelector().resourceId("org.oppia.android:id/submit_answer_b
utton")).click() //submit answer
```

Drag & Drop & Merge Interaction

```
device.findObject(UiSelector().resourceId("org.oppia.android:id/drag_drop_conte
nt_group_item")).click()

val init_position =
device.findObject(UiSelector().resourceId("org.oppia.android:id/drag_drop_item_
container").getInstance(0))

val des_position =
device.findObject(UiSelector().resourceId("org.oppia.android:id/drag_drop_item_
container").getInstance(1))

init_positon.dragTo(des_position)

device.findObject(UiSelector().resourceId("org.oppia.android:id/submit_answer_b
utton")).click() //submit answer
```

Image Region Selection Interaction.

```
val                                jupiter                                =
device.findObject(UiSelector().className("android.view.View").getInstance
(1))

jupiter.click() // and check info for jupiter

val                                saturn                                 =
device.findObject(UiSelector().className("android.view.View").getInstance
(2))

saturn.click() // to submit and exit current state
```

End Exploration or Return to Topic.

```
device.findObject(UiSelector().resourceId("org.oppia.android:id/return_to_t
opic_button")).click()
```

Note that for each window transition a wait() or performActionAndWait() is used.

PRs

Q1. Part of this project involves making changes on Oppia-web. Please take up at least one Oppia-web PR that is larger in scale and that touches the domain/controller layer.

A1. Please include the link to your PR(s) here.
Created two PRs related to controllers #12566 - Consolidate add/remove contribution rights controller handlers #12611 - Using logging.exception instead of logging.error in controllers