TESTING DOCUMENT

AR-T - AUGMENTED REALITY LEARNING APPLICATION FOR TECHNICAL GRAPHICS

STUDENT NAME: RUSSELL BRADY

STUDENT NUMBER: 15534623

SUBMISSION DATE: 19/05/2019

SUPERVISOR: MONICA WARD

TABLE OF CONTENTS

TESTING OVERVIEW

This document outlines the various types of testing which were undertaken over the course of the project. Unit testing was performed regularly as new features were being developed in the app. Integration testing began once the database and REST Api had been configured and these tests were updated when changes were made to the REST Api. UI testing was performed across the android app and web app to ensure correct functionality across both platforms. Performance and stress testing were performed to ensure the server and database could handle many users and concurrent users. User testing was conducted across several different stages which will be outlined in further detail below. The app was also verified on different devices to ensure compatibility. All of these stages are expanded on in more detail in the following sections.

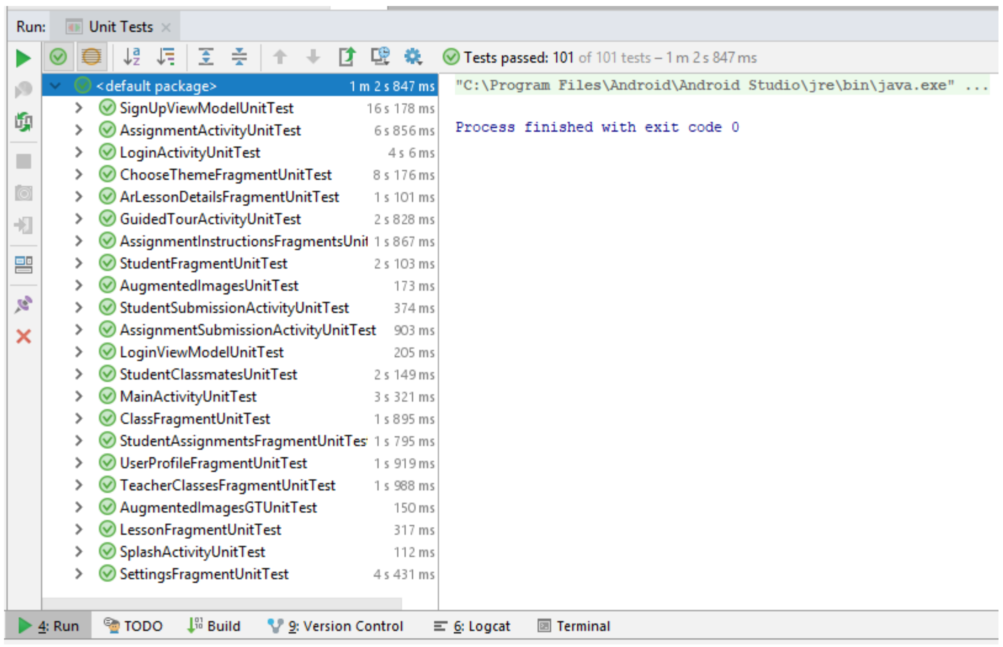
UNIT & INTEGRATION TESTING

Extensive unit testing was performed on the Android application and then integration testing was carried out on the Rest Api to test the applications routes – including database routes.

ANDROID APPLICATION

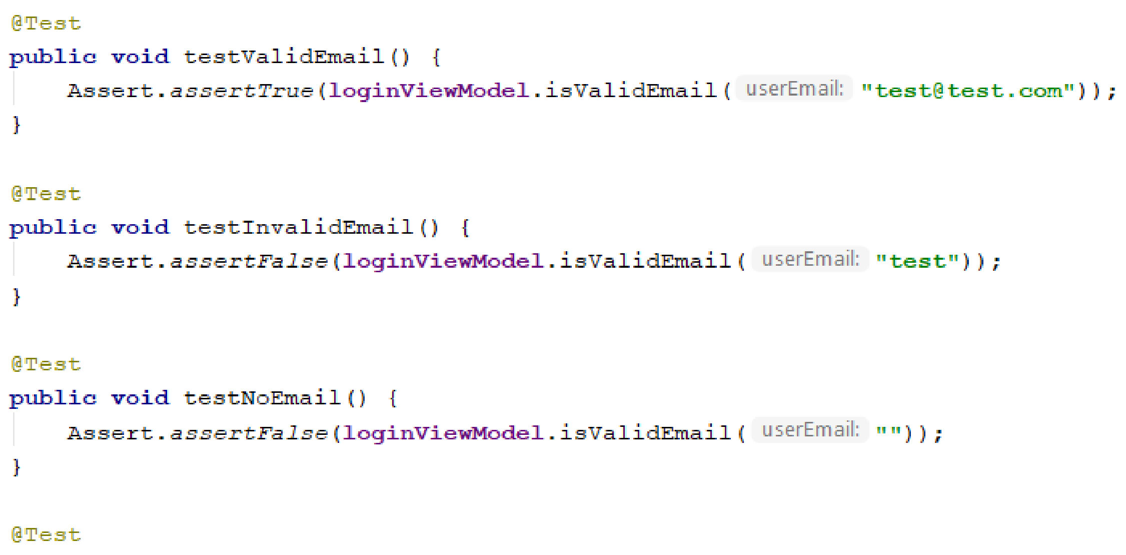
The android application was tested using the Robolectric test framework. This test framework allows for fast and reliable unit tests that run inside the JVM and don't require a device or emulator. There were two main types of unit tests conducted using this framework – Unit tests on the ViewModels which contained business logic which could be tested directly and Unit tests on the views (Activities / Fragments). These test cases were implemented over several test suites. Unit testing was conducted on a regular basis as features were being implemented. This was important as it meant these tests could be added to the CI and ran on every build to ensure tests weren't breaking as the system evolved.

These unit tests are then run together and executed by the test runner. As can be seen in the image below, there are 101 unit test cases running and passing. These test cases give class coverage of 84% and line coverage of 70%.



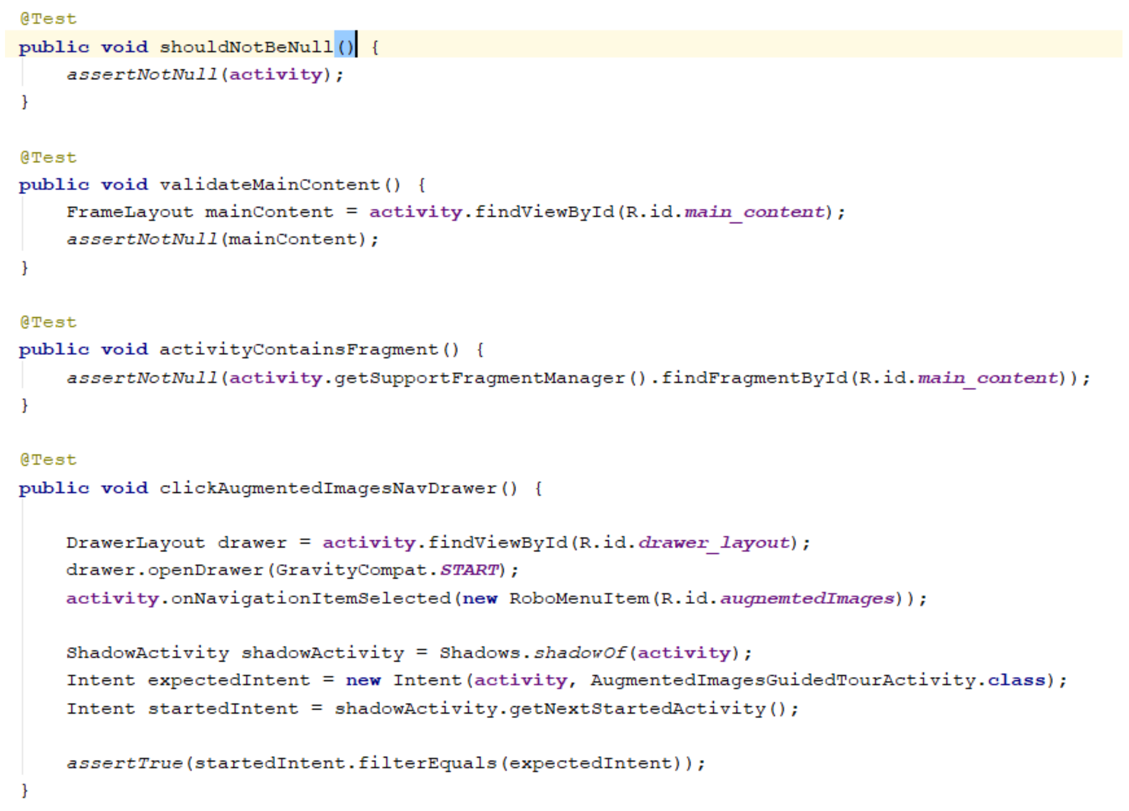
```
Tests passed: 101 of 101 tests – 1 m 2 s 847 ms

<default package>                          1 m 2 s 847 ms
  SignUpViewModelUnitTest                  16 s 178 ms
  AssignmentActivityUnitTest               6 s 856 ms
  LoginActivityUnitTest                    4 s 6 ms
  ChooseThemeFragmentUnitTest              8 s 176 ms
  ArLessonDetailsFragmentUnitTest          1 s 101 ms
  GuidedTourActivityUnitTest               2 s 828 ms
  AssignmentInstructionsFragmentsUnit      1 s 867 ms
  StudentFragmentUnitTest                  2 s 103 ms
  AugmentedImagesUnitTest                  173 ms
  StudentSubmissionActivityUnitTest        374 ms
  AssignmentSubmissionActivityUnitTest     903 ms
  LoginViewModelUnitTest                   205 ms
  StudentClassmatesUnitTest                2 s 149 ms
  MainActivityUnitTest                     3 s 321 ms
  ClassFragmentUnitTest                    1 s 895 ms
  StudentAssignmentsFragmentUnitTes        1 s 795 ms
  UserProfileFragmentUnitTest              1 s 919 ms
  TeacherClassesFragmentUnitTest           1 s 988 ms
  AugmentedImagesGTUnitTest                150 ms
  LessonFragmentUnitTest                   317 ms
  SplashActivityUnitTest                   112 ms
  SettingsFragmentUnitTest                 4 s 431 ms

"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...

Process finished with exit code 0
```

Here is an example of unit testing of one of the ViewModels business logic (LoginViewModel). In this example the function which is used to validate an email is being tested. This function must accept a certain type of string for it to be accepted as an email and the image below displays several test cases to validate the function.



```
@Test
public void testValidEmail() {
    Assert.assertTrue(loginViewModel.isValidEmail( userEmail: "test@test.com"));
}

@Test
public void testInvalidEmail() {
    Assert.assertFalse(loginViewModel.isValidEmail( userEmail: "test"));
}

@Test
public void testNoEmail() {
    Assert.assertFalse(loginViewModel.isValidEmail( userEmail: ""));
}

@Test
```

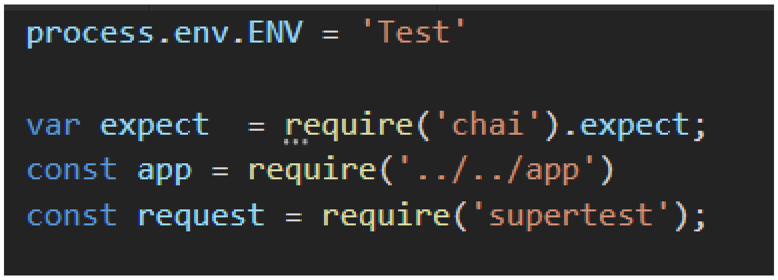
Below is an example of some of the unit tests for one of the views (MainActivity). Generally, these views are not unit testable, but Robolectric handles inflation of views, resource loading, and other tasks which are implemented in native C code on Android devices. This allows tests to do most things you could do on a real device. The first test in this suite ensures that the activity is not null upon setup, the second validates that it contains the correct view, the third ensures that the activity contains a fragment (In this case it should) and the fourth test is checking the correct behavior of clicking the main navigation drawer button in the main activity.



```
@Test
public void shouldNotBeNull() {
    assertNotNull(activity);
}

@Test
public void validateMainContent() {
    FrameLayout mainContent = activity.findViewById(R.id.main_content);
    assertNotNull(mainContent);
}

@Test
public void activityContainsFragment() {
    assertNotNull(activity.getSupportFragmentManager().findFragmentById(R.id.main_content));
}

@Test
public void clickAugmentedImagesNavDrawer() {

    DrawerLayout drawer = activity.findViewById(R.id.drawer_layout);
    drawer.openDrawer(GravityCompat.START);
    activity.onNavigationItemSelected(new RoboMenuItem(R.id.augnemtedImages));

    ShadowActivity shadowActivity = Shadows.shadowOf(activity);
    Intent expectedIntent = new Intent(activity, AugmentedImagesGuidedTourActivity.class);
    Intent startedIntent = shadowActivity.getNextStartedActivity();

    assertTrue(startedIntent.filterEquals(expectedIntent));
}
```

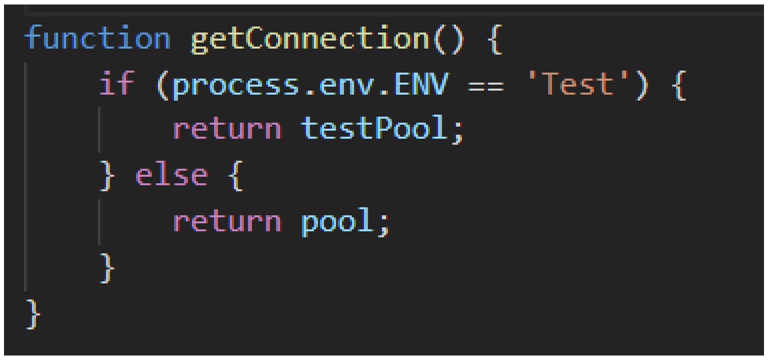
REST API

Integration testing was performed on the Node.js Express Rest Api by setting up a test database with the same schema as the real database and testing the database routes of the application. The schema of the real database was exported and imported into the test database to ensure it was an exact copy. To use the test database, the environment is changed to 'Test' during the execution of the integration test suit. This then changes the configuration of the database the Rest Api is pointing at to the test database. The environment is set in the test file:



```
process.env.ENV = 'Test'

var expect  = require('chai').expect;
const app = require('../../app')
const request = require('supertest');
```

Once this is done, when the 'app' is created, and the database is set up in the server file, the correct database is called using this code:



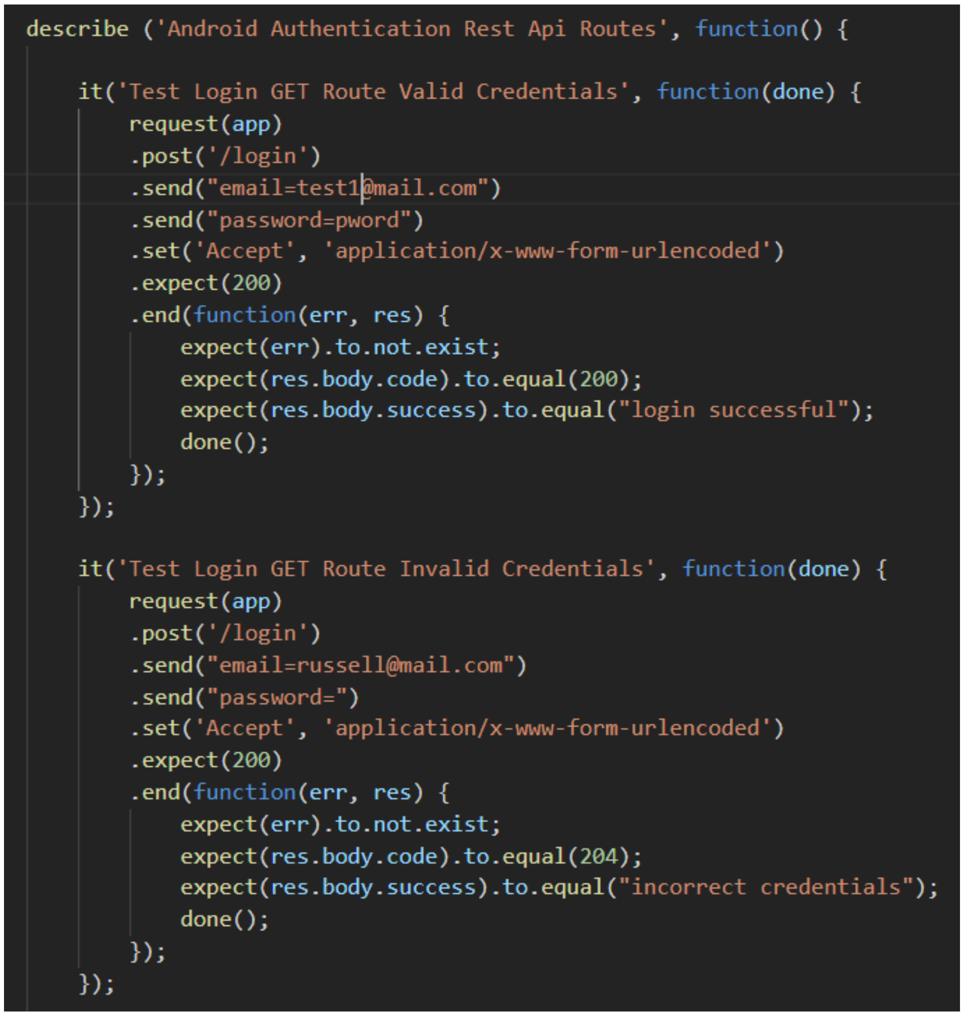
```
function getConnection() {
    if (process.env.ENV == 'Test') {
        return testPool;
    } else {
        return pool;
    }
}
```

It was very important to test using a test database as it would have been bad practice to execute test cases using the real database. The reason for this is that it may introduce redundant testing data or delete existing real user data in the tables. Setting up a test database took some extra effort, but it was worth it to ensure that the real database wasn't being contaminated with test data.

These integration tests involved testing all database routes which include:

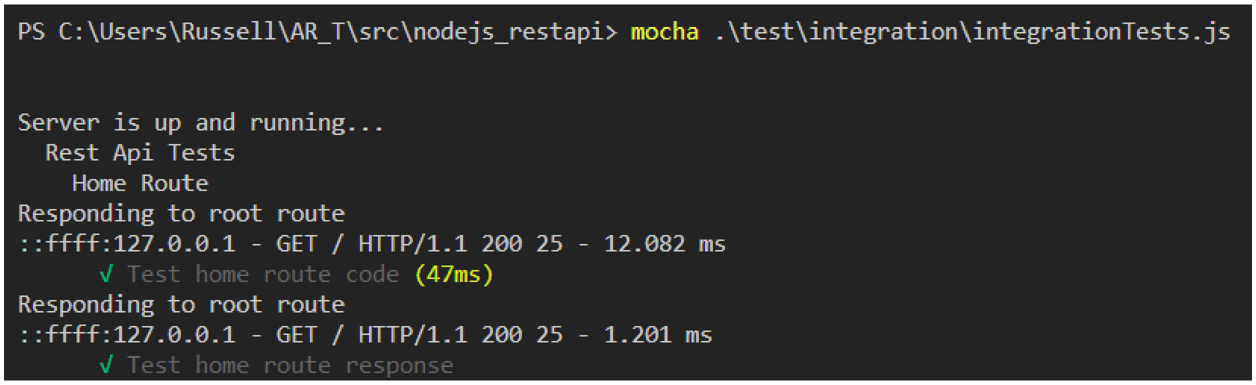
- Student / Teacher registration and login
- Web App routes
- Classroom routes (e.g. Create a class, get classes, create assignment etc.)
- Cloud Anchor routes
- 3D model routes

Below is an example of two of the integration tests which is testing the login route. The first test is validating the login route with valid credentials and the second test is validating it with invalid credentials. The test waits for the response to be returned and ensures the appropriate response is returned.

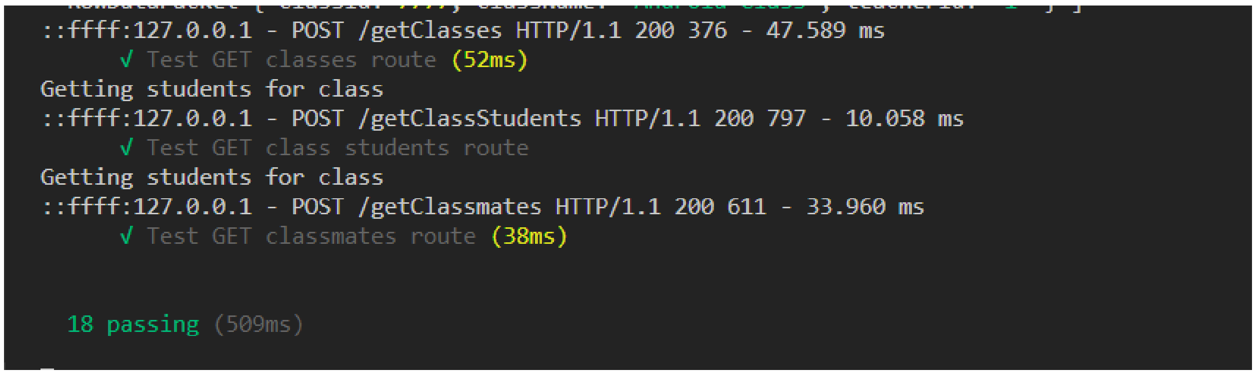


```
describe ('Android Authentication Rest Api Routes', function() {

    it('Test Login GET Route Valid Credentials', function(done) {
        request(app)
        .post('/login')
        .send("email=test1@mail.com")
        .send("password=pword")
        .set('Accept', 'application/x-www-form-urlencoded')
        .expect(200)
        .end(function(err, res) {
            expect(err).to.not.exist;
            expect(res.body.code).to.equal(200);
            expect(res.body.success).to.equal("login successful");
            done();
        });
    });

    it('Test Login GET Route Invalid Credentials', function(done) {
        request(app)
        .post('/login')
        .send("email=russell@mail.com")
        .send("password=")
        .set('Accept', 'application/x-www-form-urlencoded')
        .expect(200)
        .end(function(err, res) {
            expect(err).to.not.exist;
            expect(res.body.code).to.equal(204);
            expect(res.body.success).to.equal("incorrect credentials");
            done();
        });
    });
```

These integration tests are invaluable as they ensure the integrity of the database and the Rest Api. There are 18 integration tests running altogether.



```
PS C:\Users\Russell\AR_T\src\nodejs_restapi> mocha .\test\integration\integrationTests.js


Server is up and running...
  Rest Api Tests
    Home Route
Responding to root route
::ffff:127.0.0.1 - GET / HTTP/1.1 200 25 - 12.082 ms
      √ Test home route code (47ms)
Responding to root route
::ffff:127.0.0.1 - GET / HTTP/1.1 200 25 - 1.201 ms
      √ Test home route response
```



```
::ffff:127.0.0.1 - POST /getClasses HTTP/1.1 200 376 - 47.589 ms
      √ Test GET classes route (52ms)
Getting students for class
::ffff:127.0.0.1 - POST /getClassStudents HTTP/1.1 200 797 - 10.058 ms
      √ Test GET class students route
Getting students for class
::ffff:127.0.0.1 - POST /getClassmates HTTP/1.1 200 611 - 33.960 ms
      √ Test GET classmates route (38ms)


  18 passing (509ms)
```

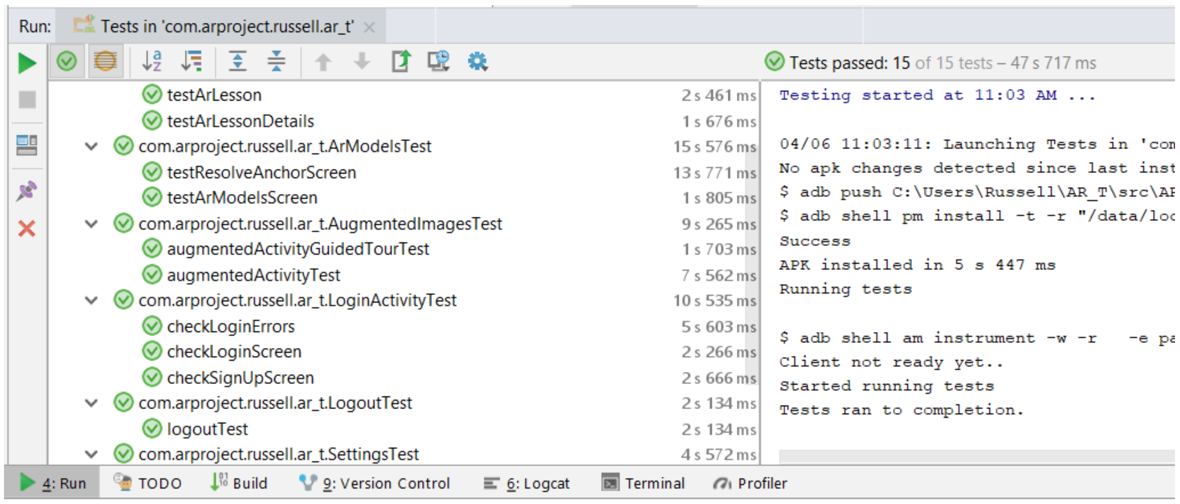
UI TESTING

ANDROID APPLICATION

In the Android application, Expresso was used as the UI testing framework. Espresso is a testing framework created by Google for Android to make it easy to write reliable user interface tests. These tests must be run on a device / emulator and are a lot slower than the unit and integration testing outlined above. However, these tests are very useful in validating the functionality of the app and that navigation through the app is correct.

There are 15 UI tests currently implemented for the Android App. These take approximately 1 minute to run. These tests validate the following functionality:

- Ensure correct application context
- Validate Login and sign up pages
- Validate main home page on login
- Using navigation bar and validating correct navigation to different screens (Settings screen etc.)
- Opening of AR screens
- Validating logout functionality



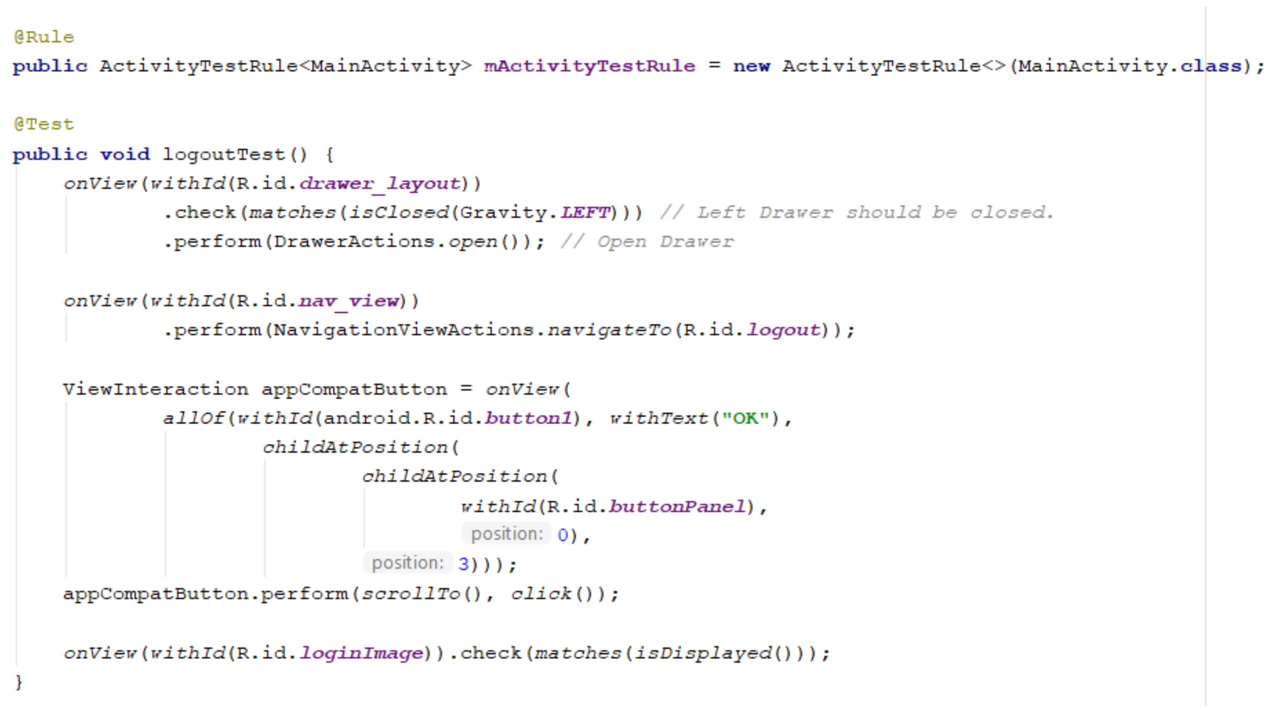
```
Tests passed: 15 of 15 tests – 47 s 717 ms

testArLesson                                      2 s 461 ms
testArLessonDetails                               1 s 676 ms
com.arproject.russell.ar_t.ArModelsTest          15 s 576 ms
  testResolveAnchorScreen                        13 s 771 ms
  testArModelsScreen                              1 s 805 ms
com.arproject.russell.ar_t.AugmentedImagesTest    9 s 265 ms
  augmentedActivityGuidedTourTest                 1 s 703 ms
  augmentedActivityTest                           7 s 562 ms
com.arproject.russell.ar_t.LoginActivityTest     10 s 535 ms
  checkLoginErrors                                5 s 603 ms
  checkLoginScreen                                2 s 266 ms
  checkSignUpScreen                               2 s 666 ms
com.arproject.russell.ar_t.LogoutTest             2 s 134 ms
  logoutTest                                      2 s 134 ms
com.arproject.russell.ar_t.SettingsTest           4 s 572 ms
```

```
Testing started at 11:03 AM ...

04/06 11:03:11: Launching Tests in 'com
No apk changes detected since last inst
$ adb push C:\Users\Russell\AR_T\src\AF
$ adb shell pm install -t -r "/data/loc
Success
APK installed in 5 s 447 ms
Running tests

$ adb shell am instrument -w -r   -e pa
Client not ready yet..
Started running tests
Tests ran to completion.
```

Here is an example of one of the defined UI tests. Before this test is ran, the main activity is opened and set up. Once the test is running it follows this sequence of steps:

- The navigation drawer of the main activity is found and opened.
- The logout button is found and clicked.
- The logout dialog appears, and the OK button is clicked.
- It is checked that the app is now in the login screen

Once this set of steps occurs, the test passes.



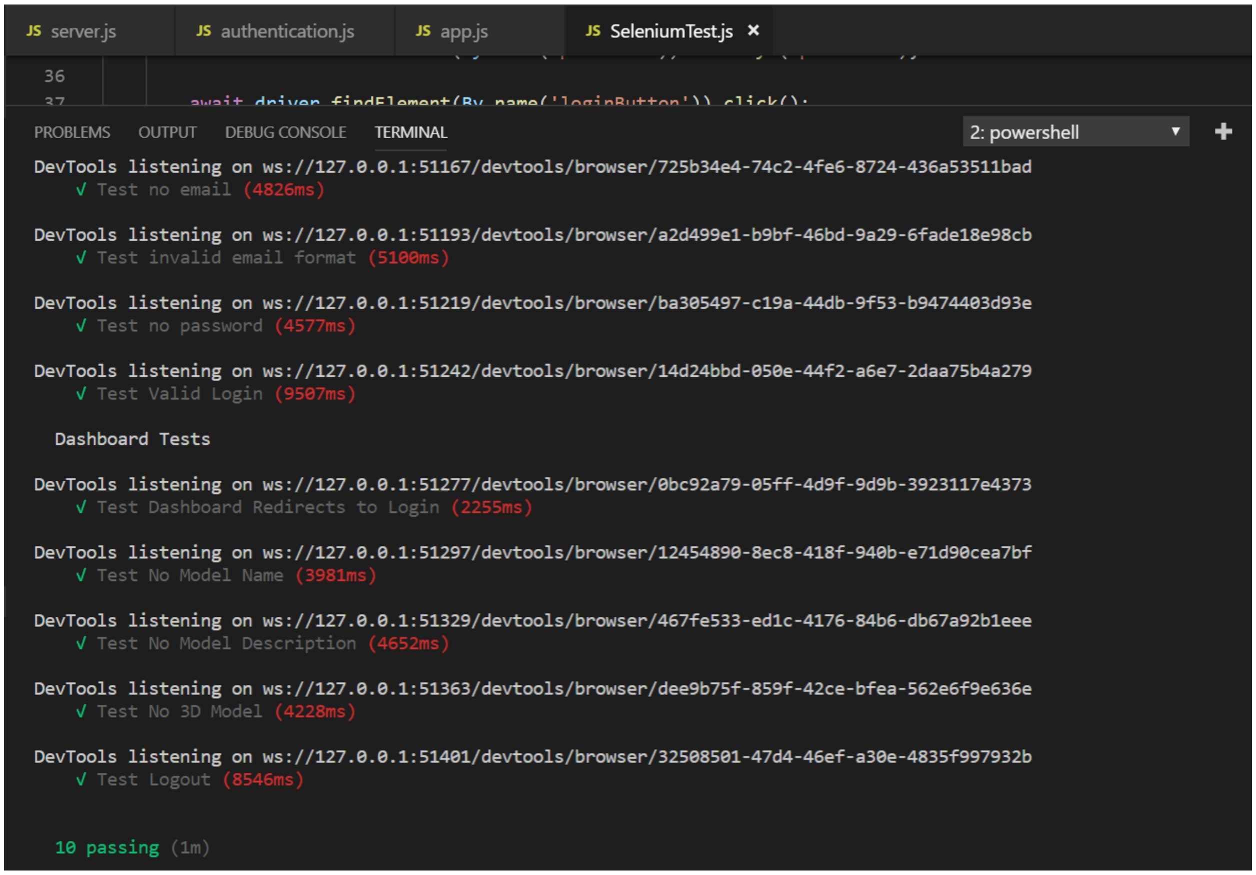
```
@Rule
public ActivityTestRule<MainActivity> mActivityTestRule = new ActivityTestRule<>(MainActivity.class);

@Test
public void logoutTest() {
    onView(withId(R.id.drawer_layout))
            .check(matches(isClosed(Gravity.LEFT))) // Left Drawer should be closed.
            .perform(DrawerActions.open()); // Open Drawer

    onView(withId(R.id.nav_view))
            .perform(NavigationViewActions.navigateTo(R.id.logout));

    ViewInteraction appCompatButton = onView(
            allOf(withId(android.R.id.button1), withText("OK"),
                    childAtPosition(
                            childAtPosition(
                                    withId(R.id.buttonPanel),
                                    position: 0),
                            position: 3)));
    appCompatButton.perform(scrollTo(), click());

    onView(withId(R.id.loginImage)).check(matches(isDisplayed()));
}
```

WEB APPLICATION

Selenium was used to UI test the web application. Selenium is a portable framework for testing web applications on a browser. Selenium was used to automate interaction with the browser in testing.

There are 10 UI tests implemented using Selenium to validate the functionality of the web application. These tests were written to validate the following functionality of the web app:

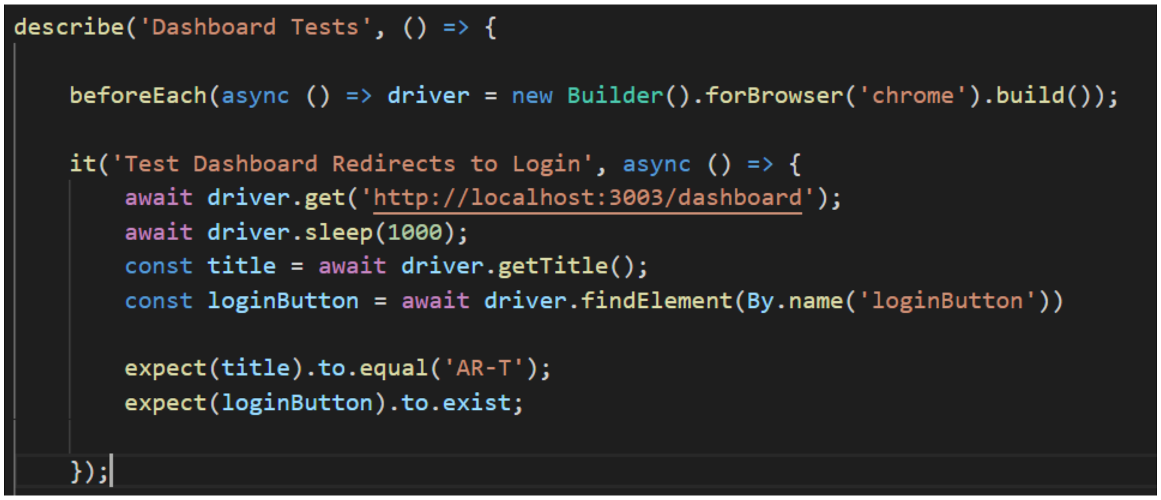
- Login – Correct / Incorrect / Missing Fields / Incorrect formats etc.
- Dashboard – Adding 3D Models – Correct / Incorrect / Missing Fields etc.
- Logout



```
DevTools listening on ws://127.0.0.1:51167/devtools/browser/725b34e4-74c2-4fe6-8724-436a53511bad
    √ Test no email (4826ms)

DevTools listening on ws://127.0.0.1:51193/devtools/browser/a2d499e1-b9bf-46bd-9a29-6fade18e98cb
    √ Test invalid email format (5100ms)

DevTools listening on ws://127.0.0.1:51219/devtools/browser/ba305497-c19a-44db-9f53-b9474403d93e
    √ Test no password (4577ms)

DevTools listening on ws://127.0.0.1:51242/devtools/browser/14d24bbd-050e-44f2-a6e7-2daa75b4a279
    √ Test Valid Login (9507ms)

  Dashboard Tests

DevTools listening on ws://127.0.0.1:51277/devtools/browser/0bc92a79-05ff-4d9f-9d9b-3923117e4373
    √ Test Dashboard Redirects to Login (2255ms)

DevTools listening on ws://127.0.0.1:51297/devtools/browser/12454890-8ec8-418f-940b-e71d90cea7bf
    √ Test No Model Name (3981ms)

DevTools listening on ws://127.0.0.1:51329/devtools/browser/467fe533-ed1c-4176-84b6-db67a92b1eee
    √ Test No Model Description (4652ms)

DevTools listening on ws://127.0.0.1:51363/devtools/browser/dee9b75f-859f-42ce-bfea-562e6f9e636e
    √ Test No 3D Model (4228ms)

DevTools listening on ws://127.0.0.1:51401/devtools/browser/32508501-47d4-46ef-a30e-4835f997932b
    √ Test Logout (8546ms)


  10 passing (1m)
```

Below is an example Selenium UI test which was written to test the dashboard route of the web app. This test validates that the user should not be able to access the dashboard before logging in. In the image below, it can also be seen that before each individual test a new Async call is made to create a new driver to run the test. What this means is that a new Chrome browser will be opened to run a clean test. After each test this driver is then torn down to kill the Chrome browser. The test follows this sequence of steps:

- Open the URL for the dashboard route
- Sleep the test for 1 second so it can be seen visually
- Get the title and login button and validate them to ensure the app has been redirected to the login screen


```
describe('Dashboard Tests', () => {

    beforeEach(async () => driver = new Builder().forBrowser('chrome').build());

    it('Test Dashboard Redirects to Login', async () => {
        await driver.get('http://localhost:3003/dashboard');
        await driver.sleep(1000);
        const title = await driver.getTitle();
        const loginButton = await driver.findElement(By.name('loginButton'))

        expect(title).to.equal('AR-T');
        expect(loginButton).to.exist;

    });
```

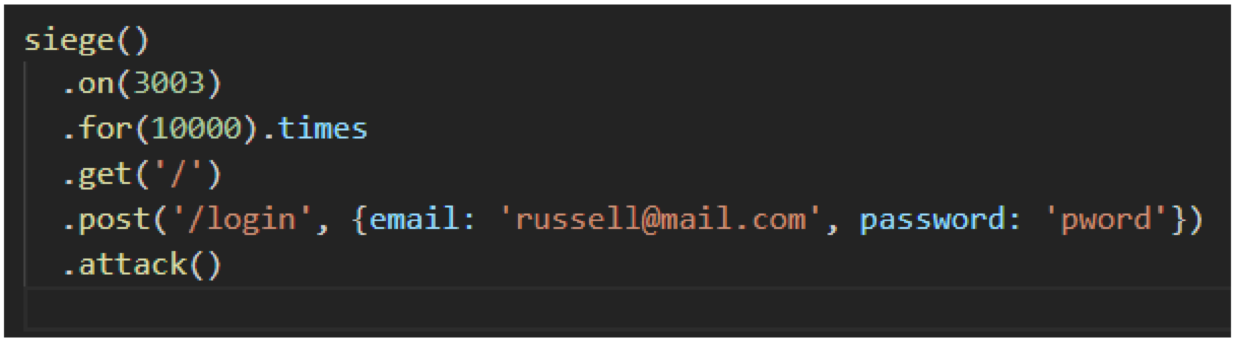
PERFORMANCE / STRESS TESTING

WEB APPLICATION & REST API

Performance / Stress testing was carried out on the application to ensure that it could handle many users at once. Load testing was carried out to test the Rest Api with a specified number of requests to test the functionality of the system under specific levels of simultaneous requests. This was done to ensure the system could handle a predefined expected volume with minimal to acceptable performance degradation. Stress testing of the Rest Api was similar only differing in that the tests are designed to increase the number of requests to the system to see how it responds. This was carried out to see at what point the performance of the system degrades or possibly fails.

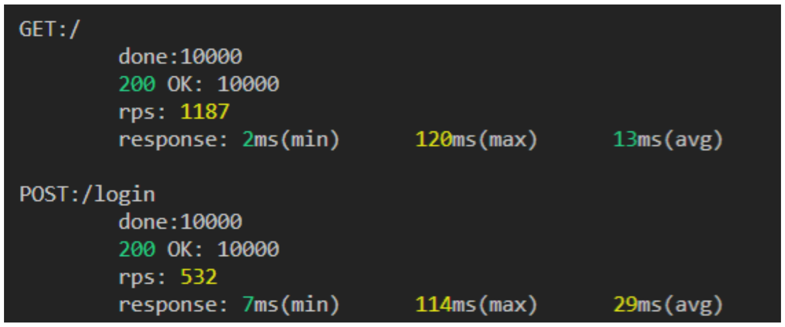
These tests were carried out just before conducting user testing to ensure that the server would be able to handle requests and wouldn't fail. Although there wasn't going to be anywhere near this number of requests, these tests were important to ensure confidence in the system and be sure that it wouldn't cause problems during user testing and the demo. The results of this testing were positive, allowing for a good level of confidence in the system.

Here is an example of load testing using Siege:


```
siege()
  .on(3003)
  .for(10000).times
  .get('/')
  .post('/login', {email: 'russell@mail.com', password: 'pword'})
  .attack()
```

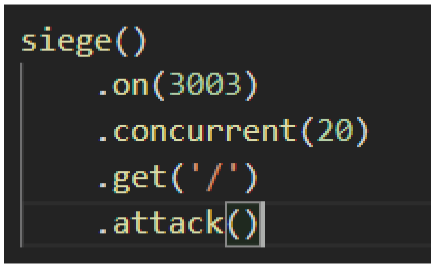
In this code, the home route of the Rest Api with 10000 requests and then doing the same with the login route, passing it valid credentials. This is designed to ensure the Rest Api can handle that number of requests, but also that the database can respond to all these requests.

The results of this test running can be seen below. There were no problems taking in this number of requests although the system did slow slightly which is to be expected.

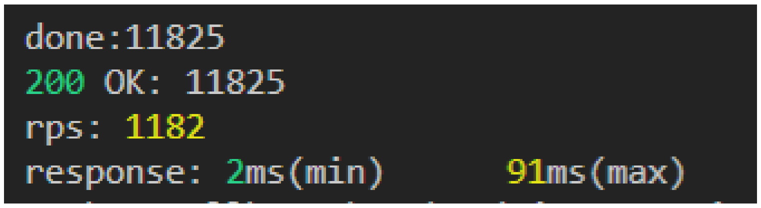


```
GET:/
        done:10000
        200 OK: 10000
        rps: 1187
        response: 2ms(min)      120ms(max)      13ms(avg)

POST:/login
        done:10000
        200 OK: 10000
        rps: 532
        response: 7ms(min)      114ms(max)      29ms(avg)
```

It was also important that checks were made to ensure the system could handle concurrent requests. Siege was again used for this. One of the routes which was tested concurrently was the home route and the command and results can be seen below. This test case ran to completion. 20 concurrent requests were specified at a time and the system was able to handle it fine.



```
siege()
    .on(3003)
    .concurrent(20)
    .get('/')
    .attack()
```



```
done:11825
200 OK: 11825
rps: 1182
response: 2ms(min)      91ms(max)
```

ANDROID APPLICATION

Stress testing of the Android app was carried out using Monkey testing to determine the robustness of the software by testing beyond the limits of normal operation.

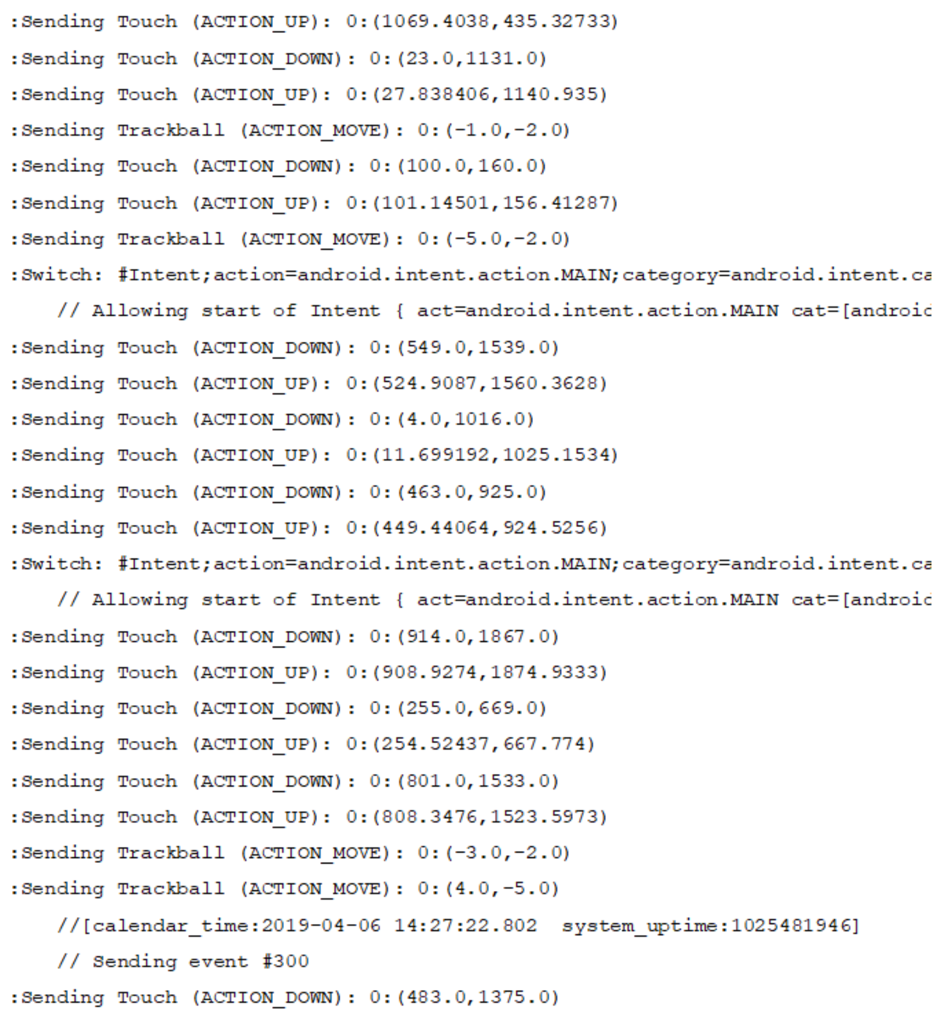
The Monkey software watches the system under test and looks for 3 main conditions:

- The Monkey watches for attempts to navigate to any other packages and blocks them.
- If the application crashes or receives any sort of unhandled exception, the Monkey will stop and report the error.
- If the application generates an application not responding error, the Monkey will stop and report the error.

This testing was performed to stress the app and find potential crashes. It was performed across the login / signup screens as well as then being logged in as a teacher and student. The monkey command was also given a seed on running each test so that the results would be reproducible should the app crash. This was the case in one instance where the app crashed in the login. The source of the crash was found in the logcat and once the problem was fixed, the monkey command was re-run with the same seed to ensure the same crash wasn't reproduced.

The Monkey program is ran using the following command - adb shell monkey -p com.arproject.russell.ar_t -v 1000 -s 2004

This command specifies the app package on the device to run the testing on as well as the number of actions that should be carried out (1000 in this case) and the seed number (2004).



```
:Sending Touch (ACTION_UP): 0:(1069.4038,435.32733)
:Sending Touch (ACTION_DOWN): 0:(23.0,1131.0)
:Sending Touch (ACTION_UP): 0:(27.838406,1140.935)
:Sending Trackball (ACTION_MOVE): 0:(-1.0,-2.0)
:Sending Touch (ACTION_DOWN): 0:(100.0,160.0)
:Sending Touch (ACTION_UP): 0:(101.14501,156.41287)
:Sending Trackball (ACTION_MOVE): 0:(-5.0,-2.0)
:Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.ca
    // Allowing start of Intent { act=android.intent.action.MAIN cat=[androic
:Sending Touch (ACTION_DOWN): 0:(549.0,1539.0)
:Sending Touch (ACTION_UP): 0:(524.9087,1560.3628)
:Sending Touch (ACTION_DOWN): 0:(4.0,1016.0)
:Sending Touch (ACTION_UP): 0:(11.699192,1025.1534)
:Sending Touch (ACTION_DOWN): 0:(463.0,925.0)
:Sending Touch (ACTION_UP): 0:(449.44064,924.5256)
:Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.ca
    // Allowing start of Intent { act=android.intent.action.MAIN cat=[androic
:Sending Touch (ACTION_DOWN): 0:(914.0,1867.0)
:Sending Touch (ACTION_UP): 0:(908.9274,1874.9333)
:Sending Touch (ACTION_DOWN): 0:(255.0,669.0)
:Sending Touch (ACTION_UP): 0:(254.52437,667.774)
:Sending Touch (ACTION_DOWN): 0:(801.0,1533.0)
:Sending Touch (ACTION_UP): 0:(808.3476,1523.5973)
:Sending Trackball (ACTION_MOVE): 0:(-3.0,-2.0)
:Sending Trackball (ACTION_MOVE): 0:(4.0,-5.0)
    //[calendar_time:2019-04-06 14:27:22.802  system_uptime:1025481946]
    // Sending event #300
:Sending Touch (ACTION_DOWN): 0:(483.0,1375.0)
```

USER TESTING

Once ethical approval was received User Testing was started. User testing was conducted across several different stages which are outlined below.

MEETING WITH EDUCATOR IN ST. PATS

The first stage of user testing involved meeting with Joe Travers, Head of the School of Inclusive and Special Education in St. Pats. The purpose of this stage of user testing was to get the opinion of a professional and to gain feedback on how best to incorporate the pedagogy element in the app. Joe introduced 3 concepts to ensure pedagogy in the app: Linguistics, Concept and Procedure. Joe stated that although there was plenty of content in the app, there was further work needed to direct this content with appropriate linguistics (Introducing language to explain the concepts being shown in the lessons) and procedure (Applying a sequence of steps). His main feedback was:

- Make no assumptions when creating lessons
- Define everything which mightn't be clear to the user
- Apply knowledge in a sequence of steps.
- Add text in the AR lessons but also allow them to be hidden
- Allow for exploratory learning
- Use prompts when the user is stuck or goes wrong
- Some form of assessment from teacher

This feedback proved invaluable and has since been implemented in the app.

LIAISING WITH T.G. TEACHERS

Another stage of user testing has been the continual liaising with two T.G. teachers. From the start of the project these two teachers have provided invaluable information in relation to what should be included in the app and what content is appropriate to includes. They have provided guidance and help in relation to forming relevant lessons in AR and how to best convey the content in the app. Both teachers reviewed the content of the app over several iterations and was able to give feedback. Some feedback included:

- AR models were too big and needed to be reduced in size.
- Don't assume prior knowledge of lesson content – explain everything
- Keep the lessons short

USER TESTING CONDUCTED IN CNOC MHUIRE GRANARD SECONDARY SCHOOL

User testing of the app was conducted in Cnoc Mhuire Granard secondary school with a T.G. teachers and five first year students. The user testing followed the following structure - The developer gave the teachers and students a brief overview of the app and its main features. Then the users testing was conducted with the teachers. Both the teacher and students were given a set of tasks to complete and following this they performed some exploratory learning through the app. The same process was then carried out with the students.

This user testing was conducted in compliance with the usability testing standard ISO 9241-11. This is an internationally recognized standard for HCI and Usability. The Usability metrics complying with this ISO standard are the following:

- Effectiveness: The accuracy with which the user completes the goals
- Efficiency: Resources expended to complete goals and time taken

- Satisfaction: The ease of user in completing goals and their happiness when using the system

Following this, the user then filled out a questionnaire as to how they found using the app. The questionnaire which was used can be found in the ethics document in the project. This provided great feedback. The feedback was also very positive.

The tasks which were carried out by the teacher included but were not limited to:

- Logging in
- Create a class
- Create an assignment
- Upload a 3D object and view it in AR using the app

Then several students from that teachers first year TG class carried out the user testing. They were also given several tasks to carry out as well as then having the chance to carry out some exploratory testing. Some of these tasks included:

- Log in
- Complete several AR lessons
- Complete a quiz based on the lessons
- Upload a 3D model and view it in AR
- Upload an assignment submission
- Use the 'Augmented Images' feature
- Change the theme colour of the app
- Log out

One piece of feedback provided by the teacher was:

"The app looks very 'Googley'. Looks and feels a lot like a standard Android app." – This feedback was positive as it meant the design principles had been followed.

One piece of feedback provided by a student was:

"The augmented reality lessons are cool. The best part is being able to make your own 3-D models and view them in augmented reality." – This feedback was very pleasing.

USER TESTING IN WEEKLY MEETING

User testing was also conducted on several occasions at the weekly meetings with the project supervisor and the other students at the meeting. This was very useful as it allowed for feedback from other students as well as from the project supervisor. This was useful as there was a constant feedback loop due allowing for issues to be found or feedback which was received to be acted upon quickly.

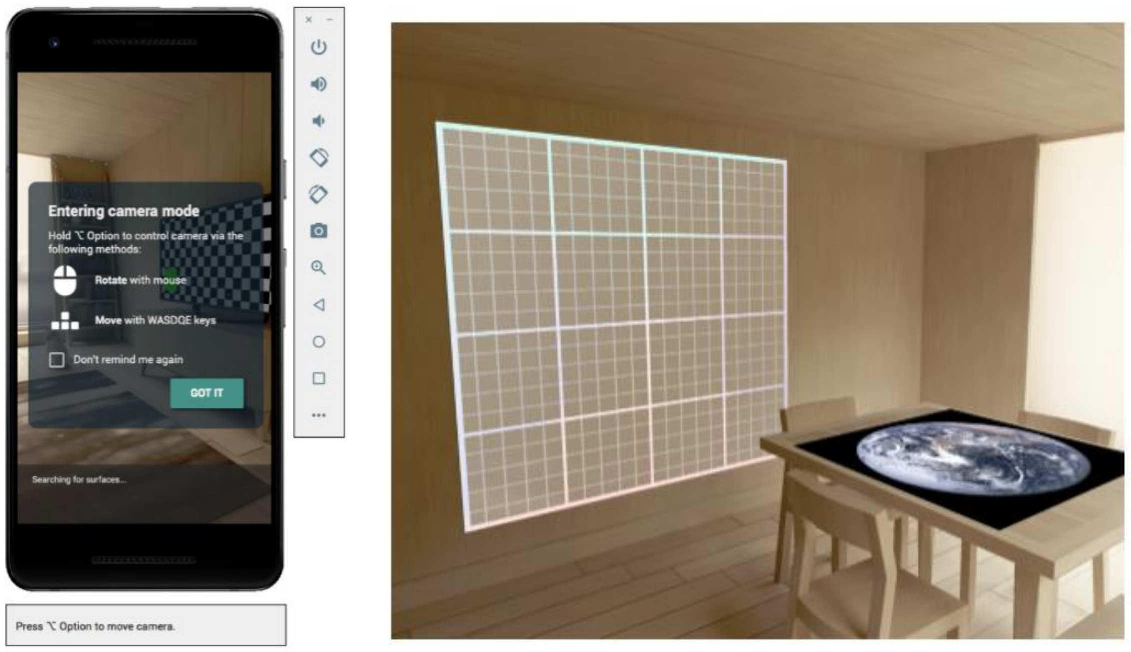
ENVIRONMENT COMPATABILITY TESTING

Environment testing involves testing the application to ensure it works in various real-world environments. In the case of this project it would mean testing it on numerous different android devices of different sizes, sdk version and device type as well as testing the web application across different browsers and ensuring correct functionality across each.

ANDROID APPLICATION

AR-T has been built to work on Android devices with a min sdk version of 24 (Nougat 7.0 – 7.1.2). No devices below this sdk version is supported and the app will not run on these devices. This is because the device hardware needed to run the Augmented Reality components of the app only exists in devices running sdk version 24 and higher.

The primary testing device was a Galaxy S7. Along with this physical device, device testing was also performed using various emulators. Some of the emulator devices which were used for testing include the Google Pixel, One Plus 5 and Galaxy A5 (2017). This ensured that the app was working correctly across a range of devices. The augmented reality components of the app could also be validated using the emulator. A virtual scene was created, as can be seen in the image below, and the augmented reality features were tested inside this virtual scene. This mimicked the behavior which would happen on the same physical device.


Entering camera mode
Hold ⌥ Option to control camera via the following methods:
Rotate with mouse
Move with WASDQE keys
Don't remind me again
GOT IT
Searching for surfaces...
Press ⌥ Option to move camera.

WEB APPLICATION

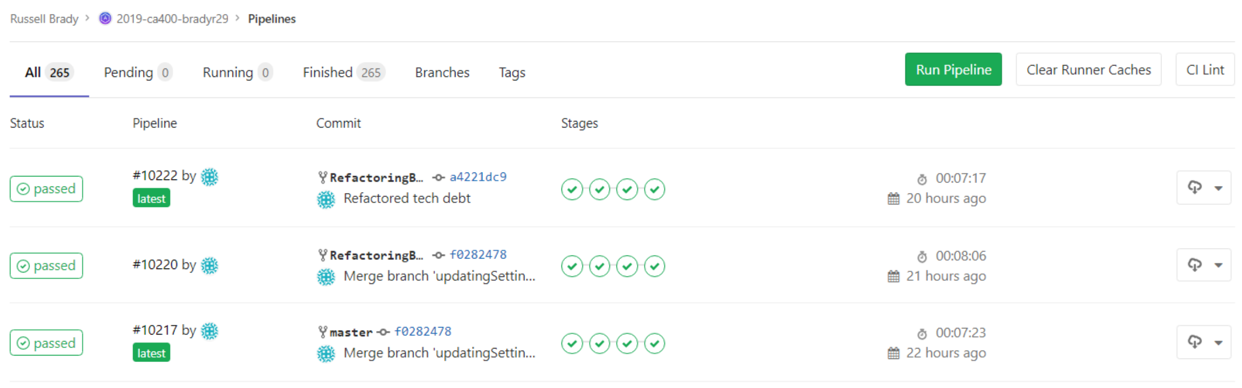
The web application was tested on Windows 10 with browsers Chrome, Firefox and MS Edge and Ubuntu 17 with Firefox and Chrome. The web application was also tested on mobile devices to ensure compatibility on smaller devices. The browsers were manually tested in a form of smoke test (Nonexhaustive set of tests that aim to ensure the most important functionality operate correctly). The criteria for acceptance was that the browser displayed normal behavior of UI elements and correct functionality of the elements.

CONTINUOUS INTEGRATION & DEVELOPMENT

Gitlab's Continuous Integration platform was implemented and used throughout the course of the project. It was a fantastic resource to have access to and it helped immensely in the success of the project. This platform was implemented in the integration and deployment of the application. The pipeline created consisted of four main stages:

- Pre – This set up the necessary tools and resources needed to run the pipeline
- Build – This ran the Android gradle build to ensure that it was passing
- Test – This stage ran Unit, Debug and Lint tests to ensure everything was passing
- Production – The app is deployed to Heroku

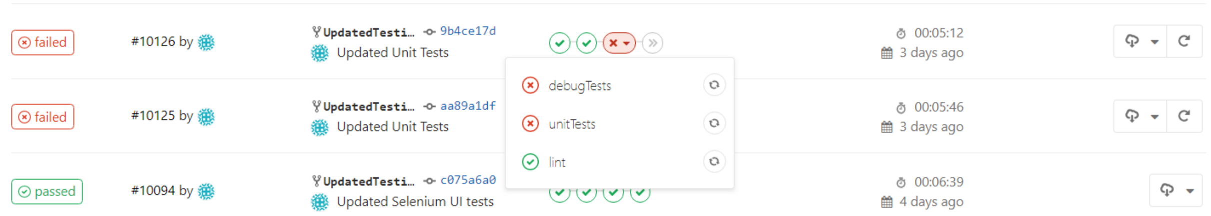
An example pipeline can be seen below:

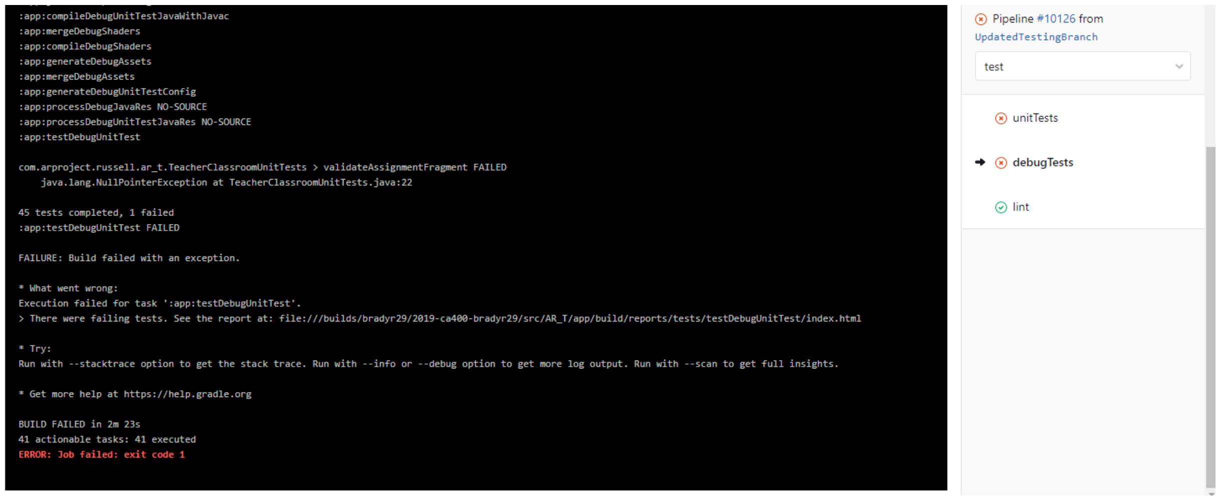


Status	Pipeline	Commit	Stages		
passed	#10222 by latest	RefactoringB... a4221dc9 Refactored tech debt	✓ ✓ ✓ ✓	00:07:17 20 hours ago	
passed	#10220 by	RefactoringB... f0282478 Merge branch 'updatingSettin...	✓ ✓ ✓ ✓	00:08:06 21 hours ago	
passed	#10217 by latest	master f0282478 Merge branch 'updatingSettin...	✓ ✓ ✓ ✓	00:07:23 22 hours ago	

The CI ran every time a commit was pushed to a branch or a branch was created / merged. This was very useful as all the development was carried out on branches created from Master. Developing on branches and running the CI before creating a merge request into Master allowed for testing of the application before a feature was added to the app. This meant bugs were found and could be fixed while preventing their introduction to the code in Master. Following this process allowed for the detection and location of bugs / failures quickly, as the problem could be backtracked to a code change in a commit on a certain branch.

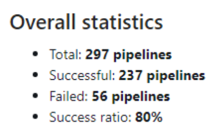
The image below displays two failed branches. Looking at the log of the CI build below, one test (ValidateAssignmentFragment in TeacherClassroomUnitTests) was failing the build. The problem was found and solved quickly. This is one example of how the CI greatly benefited the testing process, and the development process in general. There have been over 50 branches successfully merged into Master following successful CI builds.





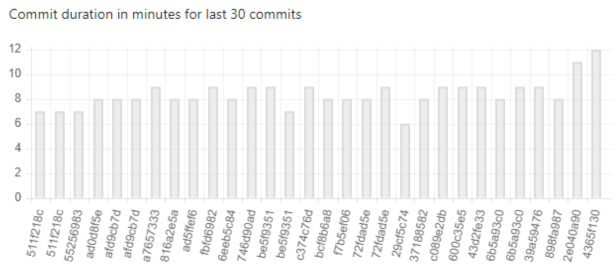
```
:app:compileDebugUnitTestJavaWithJavac
:app:mergeDebugShaders
:app:compileDebugShaders
:app:generateDebugAssets
:app:mergeDebugAssets
:app:generateDebugUnitTestConfig
:app:processDebugJavaRes NO-SOURCE
:app:processDebugUnitTestJavaRes NO-SOURCE
:app:testDebugUnitTest

com.arproject.russell.ar_t.TeacherClassroomUnitTests > validateAssignmentFragment FAILED
    java.lang.NullPointerException at TeacherClassroomUnitTests.java:22

45 tests completed, 1 failed
:app:testDebugUnitTest FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':app:testDebugUnitTest'.
> There were failing tests. See the report at: file:///builds/bradyr29/2019-ca400-bradyr29/src/AR_T/app/build/reports/tests/testDebugUnitTest/index.html

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output. Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 2m 23s
41 actionable tasks: 41 executed
ERROR: Job failed: exit code 1
```

The overall statistics for the CI can be seen below along with a graphic of all pipelines which ran over the course of the project. The green represents passing pipelines and the grey represents failing pipelines.

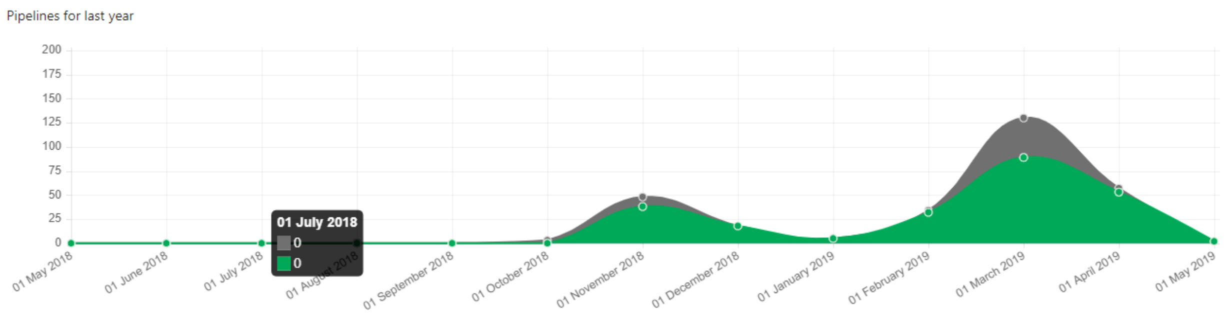


Overall statistics

- Total: **297 pipelines**
- Successful: **237 pipelines**
- Failed: **56 pipelines**
- Success ratio: **80%**



Commit duration in minutes for last 30 commits
511f218c
511f218c
ad0d8f5e
afd9cb7d
afd9cb7d
a7657333
816a2e5a
ad5ffef6
fbfd6982
6eeb5c84
746d90ad
be5f9351
be5f9351
c374c76d
bcf8b6a8
f7b5ef06
72fdad5e
72fdad5e
29cf5c74
c089e2db
600c35e5
43d2fe33
6b5a93c0
6b5a93c0
39a59476
898fa987
2e040a90
4365f130



Pipelines for last year
01 July 2018
01 May 2018
01 June 2018
01 July 2018
01 August 2018
01 September 2018
01 October 2018
01 November 2018
01 December 2018
01 January 2019
01 February 2019
01 March 2019
01 April 2019
01 May 2019