COMP6239 Coursework Report
Education Support App

Group 01
Dimitar Tsvetkov - 30771617 (ddt1u19@soton.ac.uk)
Leonid Goldberg - 30211743 (lg1n22@soton.ac.uk)
Mihaela Florea - 30838215 (mf2u19@soton.ac.uk)
Tom Hoad - 30914515 (tdh1g19@soton.ac.uk)

May 2023

COMP6239 Coursework - Proposed Marks Distribution

To be submitted as a part of your report submission
Record here your proposed distribution of the total number of marks awarded to your team.

Please enter all requested details and the percentage of the **total** team effort contributed by each team member. The contribution percentages **must** total 100%.

Each team member **must** sign and date the form before submission to confirm that they **agree** with the **proposed** distribution.
Only one fully completed form per team is necessary (but see below when this is not possible), to be submitted as the first page of your team report.
Your proposal will normally be followed but other evidence such as *logbooks* may be considered. **The course leader's decision is final.**

Group:	01

Student ID	Email ID	Percentage	Signature	Date
30771617	ddt1u19	25	Dimitar Tsvetkov	18 May 2023
30914515	tdh1g19	25	Tom Hoad	18 May 2023
30211743	lg1n22	25	Leonid Goldberg	18 May 2023
30838215	mf2u19	25	Mihaela Florea	18 May 2023

If your team is unable to agree on a proposed distribution, please talk to the module leader before submitting your form. Your team should **avoid this,** if at all possible, because it will be interpreted as demonstrating a general lack of competence.

Contents

Introduction

The aim of this project is to develop a mobile application for Education Support in which Educators can introduce Courses, with several Activities in the form of multiple-choice quizzes and Learners can join Courses and complete Activities, while the Educators can track their progress.

Requirements

ID	Title	Description
F1	Log In	The system should allow users to create an account with an email address and password, and select whether they are a Learner or an Educator
F2	My Courses and Explore Courses	The system should let users explore the list of available courses and the list of courses each educator and learner is signed up for/created.
F3	Create Course	The system should allow Educators to create Courses with a name and description
F4	Create Activities	The system should allow Educators to create for each Course one or more Activities in the form of a multiple-choice quiz. The Educator should be able to give the Activity a name.
F5	Create Questions	The system should allow Educators to create for each Activity one or more Questions. Each Question should have – the Question text – a number of possible Answer texts – one or more correct Answers
F6	Learners Join Courses	The system should allow Educators to add Learners to Courses they have created, by the Learner's email address. Also, Learners may choose to join one or more Courses from a list of all Courses.
F7	Complete Activities	The system should allow Learners to see a list of the Courses they are signed up to, along with all Activities within those Courses and complete Activities. Learners should be able to view each Activity's Questions one at a time and must choose one or more Answers
F8	Show Results	The system should allow the Learner to see, at the end of an Activity, how many Questions they answered correctly as well as to review the Questions and see whether they have chosen the correct Answer(s) for each question. The system should show the appropriate Answer(s) if they did not.

F9	Statistics	The system should allow Educators to see statistics of how Learners have performed in an Activity (i.e. how many Questions each Learner has answered correctly)
F10	Account Page	The system should allow users to view their account information and personalise their details. This includes profile picture, name and biography.
F11	Database	The system should store user, course, activity, questions data and more within a database.

Design

Figma Mockups

For some of our more important app pages, we decided on creating some mockup designs using the Figma UI design tool [5]. This tool was very beneficial because it is designed to provide collaborative tools so we could all work at the same time.

We designed the login page and the activities menu pages. The design of the login is an essential part of the user experience being the first thing the user sees. The activities menu meanwhile is copied across the explore courses and my courses menu pages so this design needed to be solid.

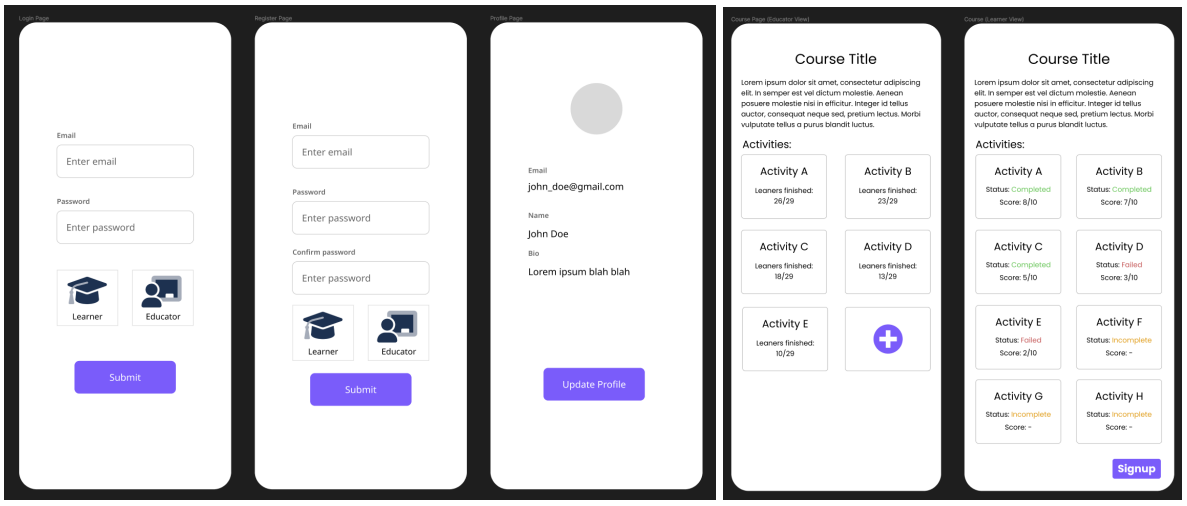

Figure 1: Figma Mockups

Our initial designs focus on a grid layout for our menus, without any header or navbar to clear up space. Our final design encountered some issues with spacing with this design so we later switched to a layout of rows.

Activity Diagram

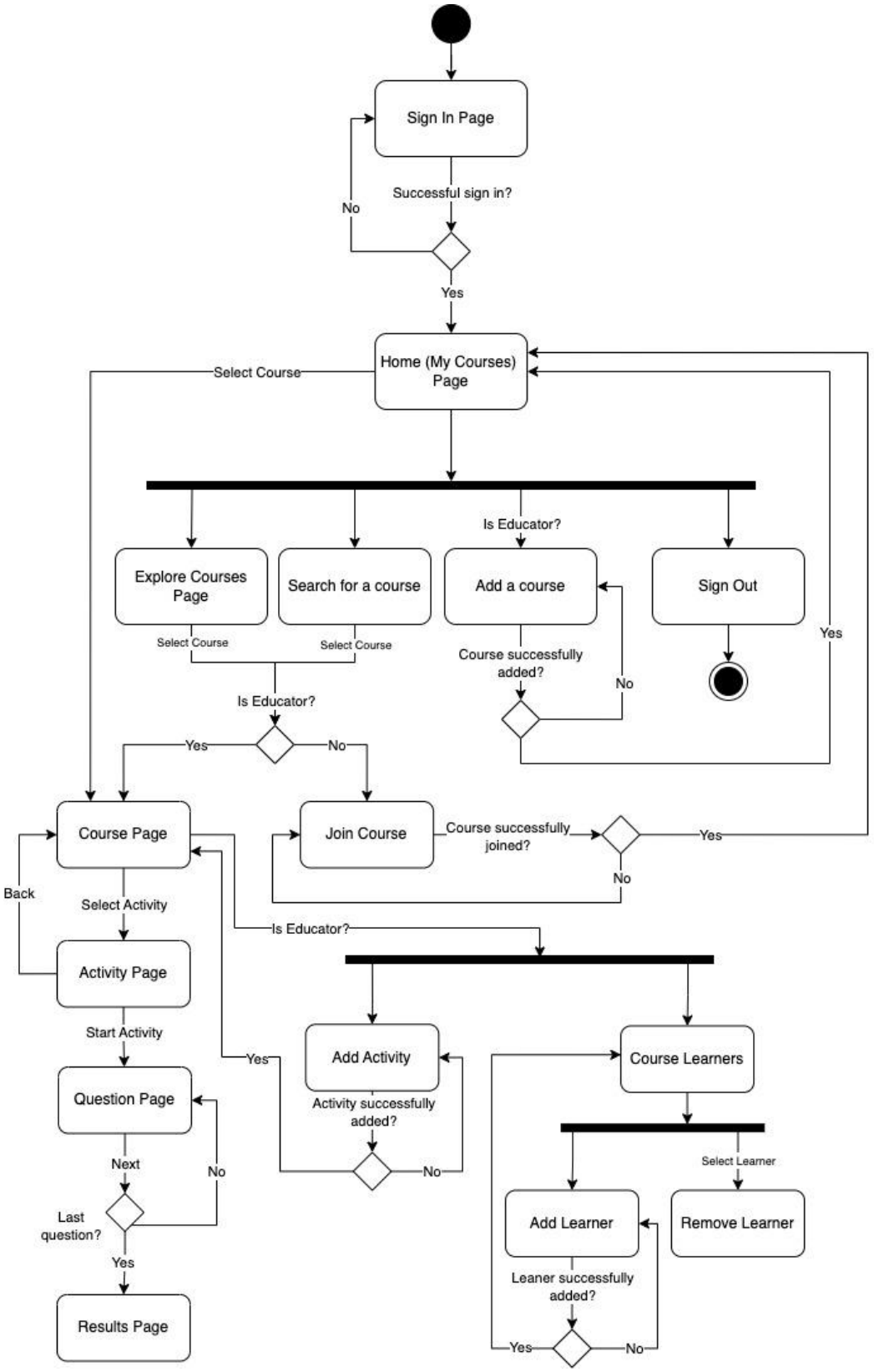

Figure 2: Activity Diagram

Database Design

Figure 3: Database ERD diagram

Implementation

Project Setup

Jira

For project management we have used agile methodology. To facilitate this we have used the Jira Software tool [1]. This has been an efficient system since we were able to report bugs and it made it easy for us to understand what each team member was working on, and what they might need help with.

GitHub

To work collaboratively, we set up a group GitHub repository to host our work. Our initial methodology was to have each person target a specific part of the app, but we found that later on we would encounter some definite crossover so it was important using pull requests for big features whilst leaving proper commenting and descriptions for people to pick up work where others left it.

Frontend

Log In Page

The “Log In” page (F1) is the first screen that the users see when the app is opened. To sign in, the user has to enter an email address and a password then select the “Educator” or “Learner” button depending on which account type they want to create. If that user email already exists in the database, they will be logged in, otherwise a new account would be created for them followed by automatic login. This removed the extra step of users having to perform the registration and log in separately.

```
auth.createUserWithEmailAndPassword(email, password)
    .addOnCompleteListener(this.requireActivity()) { task ->
        if (task.isSuccessful) {
            // Sign in success, update UI with the signed-in user's information
            // Add user to Realtime db
            addUser(User(email, educator = educator), auth.uid,password)

            findNavController().navigate(
                R.id.action_navigation_loginFragment_to_navigation_mycoursesFragment
            )
        } else {
            // If sign in fails, display a message to the user.
            // If it's because email already exists then log in
            if (task.exception is FirebaseAuthUserCollisionException) {
                logIn(email, password)
            } else {
                Toast.makeText(
                    this.requireContext(), text: "Registration failed.",
                    Toast.LENGTH_SHORT
                ).show()
            }
        }
    }
```

Figure 4: Registration code
File path: app/src/main/java/com/example/educationsupport/ui/login/LoginFragment.kt

My Courses and Explore Menus

The two main pages of our app are the “My Courses” and “Explore Courses” pages (F2). Once a user has logged in, they are launched onto the my courses page and can use a bottom navbar to switch between these two. Both pages share a very similar design layout to each other, but have some small differences depending on the page and whether the user is an educator or learner.

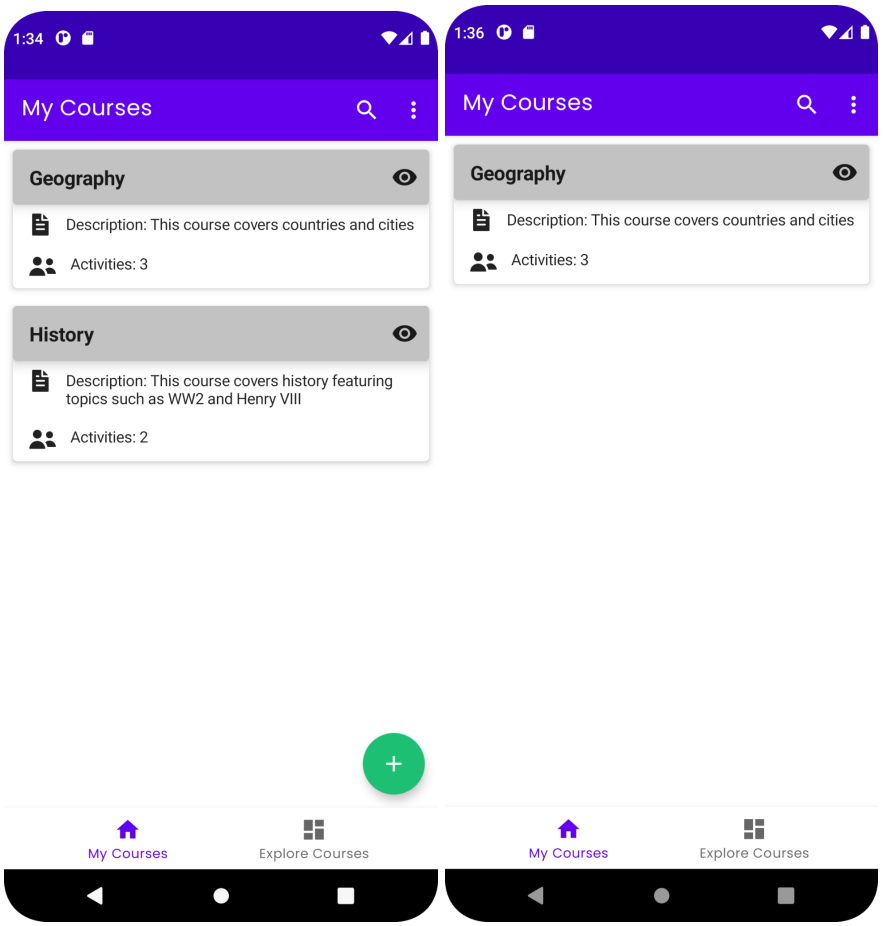

Figure 5: My Courses (L - Educator, R - Learner)

```
Row { this: RowScope
    Icon(
        painter = painterResource(id = R.drawable.description),
        contentDescription = "Description icon",
        modifier = Modifier
            .padding(start = 15.dp, end = 5.dp, top = 8.dp, bottom = 8.dp)
            .size(20.dp)
    )
    Text(
        text: "Description: ${course.second.description}",
        style = MaterialTheme.typography.subtitle2,
        modifier = Modifier
            .padding(start = 8.dp, top = 8.dp, bottom = 8.dp)
            .fillMaxSize()
    )
}
```

Figure 6: Example of using Jetpack Compose for My Courses page

Both of these pages rely heavily on Jetpack Compose. Given that we have multiple courses and need to display them all as a list, Compose was very useful for creating this template of reusable UI elements. Each course is given a title, description and learners finished count directly on the tab. The tab is detailed with icons to indicate function better.

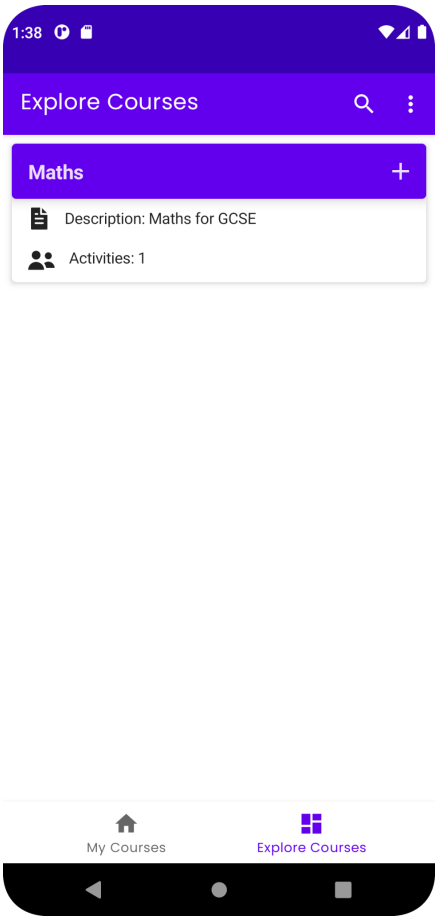

Figure 7: Explore Courses Page

In addition to this, we have also implemented a floating action button that allows educators to create new courses (F3) and a header with relevant functions such as search, profile and signout. The floating action button is an extension of our initial design by putting this functionality in the easiest accessible point of the screen and is interactable at all times for educators. As for our additional header functionalities, these were designed to give the learner and educators more control over their experience than just scrolling.

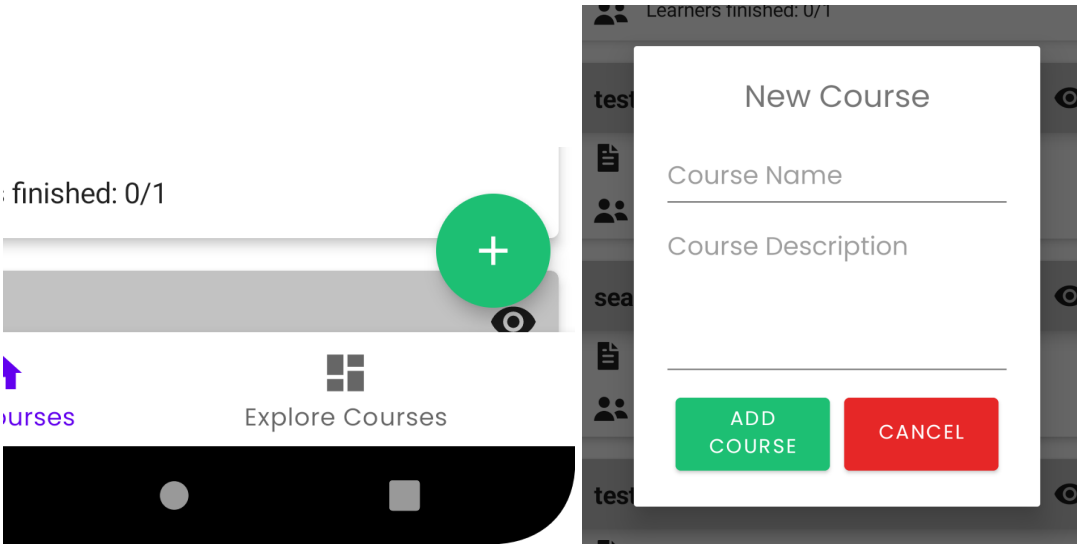

Figure 8: The FAB displayed to educators to add a new course and the dialog that appears when selected.

```
/**
 * Creates a button for adding a new course
 */
Dimitar Tsvetkov +1
private fun createAddNewCourseButton() {
    if (user.educator) {
        binding.newCourseButton.visibility = View.VISIBLE
        binding.newCourseButton.setOnClickListener { it: View!
            val dialog = Dialog(this.requireContext())
            dialog.setContentView(R.layout.dialog_new_course)
            dialog.findViewById<Button>(R.id.add_course_button).setOnClickListener { it: View!
                val courseName = dialog.findViewById<EditText>(R.id.courseName).text
                val courseDescription =
                    dialog.findViewById<EditText>(R.id.courseDescription).text
                if (courseName.toString().isNotEmpty() && courseDescription.toString()
                        .isNotEmpty()
                ) {
                    // adds course and refreshes data again after successful add
                    addCourse(
                        Course(courseName.toString(), courseDescription.toString(), auth.uid!!),
                        auth.uid!!,
                        user
                    )
                    dialog.dismiss()
                }
            }
            dialog.findViewById<Button>(R.id.cancel_course_button).setOnClickListener { it: View!
                dialog.dismiss()
            }
            dialog.show()
        }
    }
}
```

Figure 9-a: Code for creating a new course.
Filepath: app/src/main/java/com/example/educationsupport/ui/courses/MyCoursesFragment.kt

```kotlin
/**
 * Adds a course to the database
 */
Dimitar Tsvetkov
private fun addCourse(course: Course, userID: String, user: User) {
    // Generating a new courseId
    val courseID = coursesTable.push().key
    if (courseID != null) {
        // Adding new course to the database
        coursesTable.child(courseID).setValue(course)
            .addOnSuccessListener { it: Void!
                user.courseIDs[courseID] = true
                // updating user course ids
                usersTable.child(userID).setValue(user)
                    .addOnSuccessListener { it: Void!
                        getUserCoursesFromDB(user.courseIDs.keys)
                        getUserDataFromDB()
                    }
            }
    }
}
```

Figure 9-b: Code for creating a new course.
Filepath: app/src/main/java/com/example/educationsupport/ui/courses/MyCoursesFragment.kt

Each card representing a course can be tapped to open the specific course page. For the explore course page, we provide the learner the functionality to sign up to a course (F6). When a learner has signed up, that particular course will disappear from the explore page and appear on the my courses page.

Courses

For each course, we provide a similar UI to the “My courses” page but with activities instead of courses. These activity cards use the same compose generation and also can be searched.

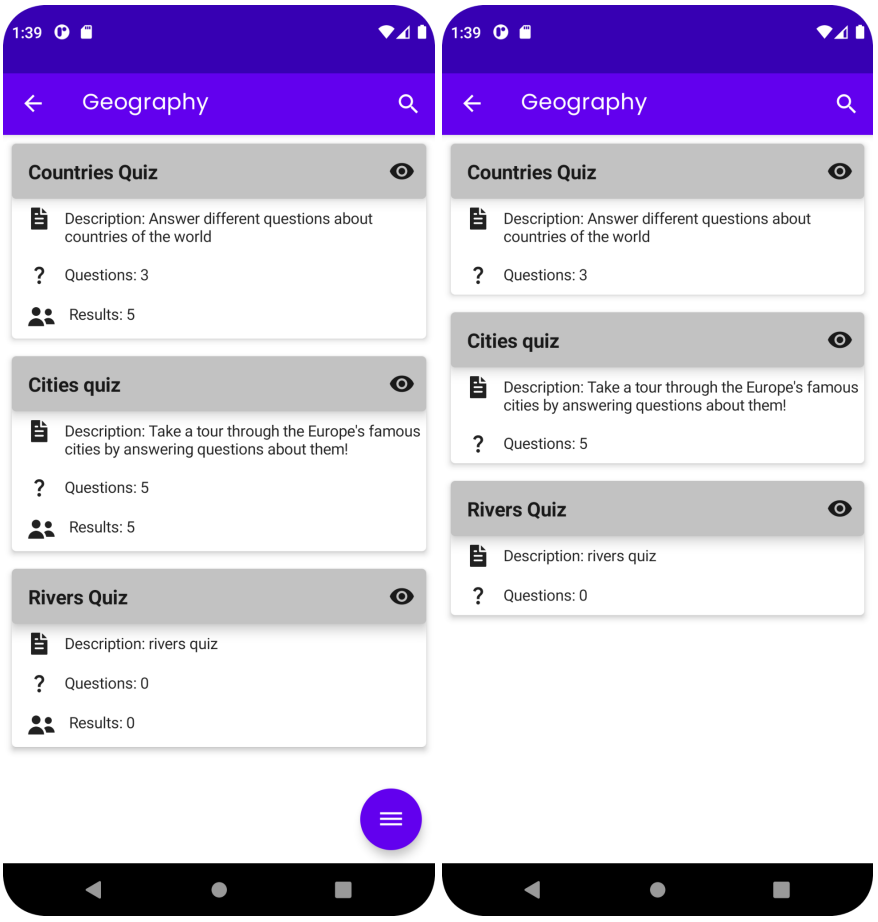

Figure 10: Course Page (L - Educator, R - Learner)

We provide the number of questions for any user type to see, whilst educators can also see the number of results submitted. The biggest difference on this page is an extended floating action button that allows new activities (F4) to be created as well as adding learners by email to the course (F6). This functionality is exclusive to educators, but is designed to be always accessible without being too obtrusive.

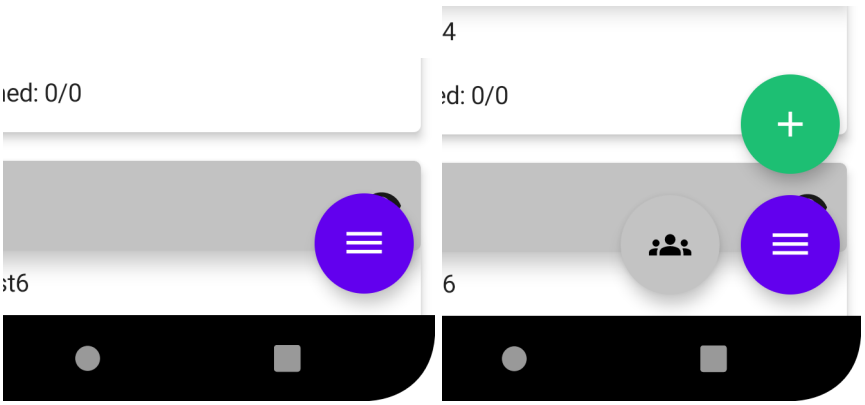

Figure 11: The expandable FAB displayed to educators when viewing their courses.The green button can add a new activity and the grey button opens the fragment to add a new learner.

```kotlin
/**
 * Applies listeners to the expandable FAB
 */
Tom Hoad +1
private fun createAddNewActivityButton() {
    if (educator) {
        binding.openFabsButton.visibility = View.VISIBLE

        binding.openFabsButton.setOnClickListener { it: View!
            if (binding.newActivityButton.visibility == View.INVISIBLE) {
                binding.newActivityButton.visibility = View.VISIBLE
                binding.showLearnersButton.visibility = View.VISIBLE
            } else {
                binding.newActivityButton.visibility = View.INVISIBLE
                binding.showLearnersButton.visibility = View.INVISIBLE
            }
        }

        // Add new activities button
        binding.newActivityButton.setOnClickListener { it: View!
            val dialog = Dialog(this.requireContext())
            dialog.setContentView(R.layout.dialog_new_activity)

            dialog.findViewById<Button>(R.id.add_activity_button).setOnClickListener { it: View!
                val activityName = dialog.findViewById<EditText>(R.id.activityName).text
                val activityDescription = dialog.findViewById<EditText>(R.id.activityDescription).text

                if (activityName.toString().isNotEmpty() && activityDescription.toString().isNotEmpty()) {
                    // adds activity and refreshes data again after successful add
                    addActivity(Activity(courseId, activityName.toString(), activityDescription.toString()), courseId, course)
                    dialog.dismiss()
                }
            }
            dialog.findViewById<Button>(R.id.cancel_activity_button).setOnClickListener { it: View!
                dialog.dismiss()
            }
            dialog.show()
        }

        // Adds a learner to the course
        binding.showLearnersButton.setOnClickListener { it: View!
            val bundle = Bundle()
            bundle.putString("courseId", courseId)
            bundle.putSerializable("course", course)
            findNavController().navigate(
                R.id.action_navigation_courseFragment_to_courseLearnersFragment,
                bundle
            )
        }
    }
}
```

Figure 12: Code for creating new activity button

Filepath: app/src/main/java/com/example/educationsupport/ui/courses/CourseFragment.kt

```kotlin
/**
 * Adds an activity to the database
 *
 * @param activity The new activity
 * @param courseId The current course id
 * @param course The current course
 */
Tom Hoad
private fun addActivity(activity: Activity, courseId: String, course: Course) {
    // Generating a new activityId
    val activityId = activitiesTable.push().key
    if (activityId != null) {
        // Adding new activity to the database
        activitiesTable.child(activityId).setValue(activity)
            .addOnSuccessListener { it: Void!
                course.activityIds[activityId] = true
                // updating course activity ids
                coursesTable.child(courseId).setValue(course)
                    .addOnSuccessListener { it: Void!
                        getCourseActivitiesFromDB(course.activityIds.keys)
                    }
            }
    }
}
```

Figure 13: Adding new activity code
Filepath: app/src/main/java/com/example/educationsupport/ui/courses/CourseFragment.kt

To add learners to a course, a separate menu is accessible. This menu shows a full list of learners who are signed up and a floating action button provides the ability to add new learners. Selecting a learner from this list will produce a popup to remove that learner.

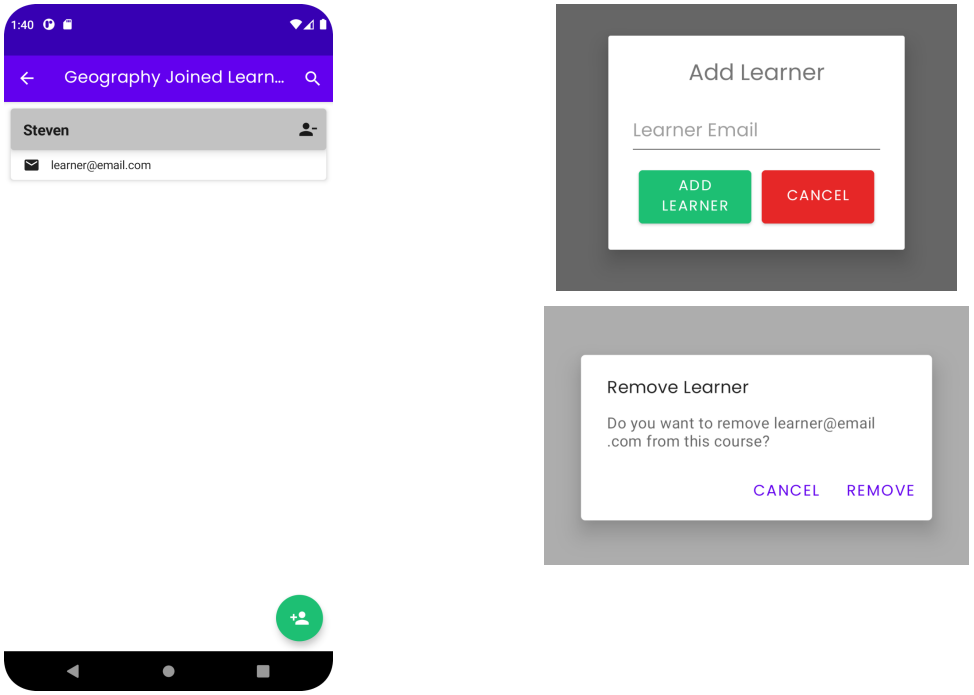

Figure 14: The page displaying the list of enrolled learners on a particular course. The FAB provides a dialog to add a new learner.

```kotlin
/**
 * Updates the current course in the database
 *
 * @param courseId The current course id
 * @param course The current course
 * @param email The new learner email
 */
Dimitar Tsvetkov
private fun addUserToCourse(courseId: String, course: Course, email: String) {
    // Gets the user with the matching email from the db
    usersTable.addListenerForSingleValueEvent(object : ValueEventListener {
        override fun onDataChange(dataSnapshot: DataSnapshot) {
            for (dataSnap in dataSnapshot.children) {
                val user = dataSnap.getValue(User::class.java)!!
                if (user.email == email) {
                    course.learnerIds[dataSnap.key.toString()] = true
                    user.courseIDs[courseId] = true

                    // Updates the course with the new learner id
                    coursesTable.child(courseId).setValue(course)

                    // Updates the user with the new course id and refreshes list
                    usersTable.child(dataSnap.key.toString()).setValue(user).addOnSuccessListener { it: Void!
                        getLearnersDataFromDB()
                    }
                    break
                }
            }
        }

        override fun onCancelled(databaseError: DatabaseError) {}
    })
}
```

Figure 15: Adding User to a Course
Filepath: app/src/main/java/com/example/educationsupport/ui/courses/CourseLearners.kt

Similar to the previous menu, each activity card will either load the quiz or an edit questions menu depending on if the user is a learner or educator.

Creating Questions

In the list of courses, the educator can click on a course they own and see a list of questions in that course. If they click on a specific question, they will be able to edit that question by changing its title and answer options. There is a button to either add a new multiple choice option, or remove the last option in the multiple choice answer set. Educators can indicate whether a specific option is correct by editing the checkmark next to the option.

Clicking on the green circular “+” button in the corner of the screen will create a new question. The screen for creating a new question is the same as for editing an existing one shown on the right of the figure below.

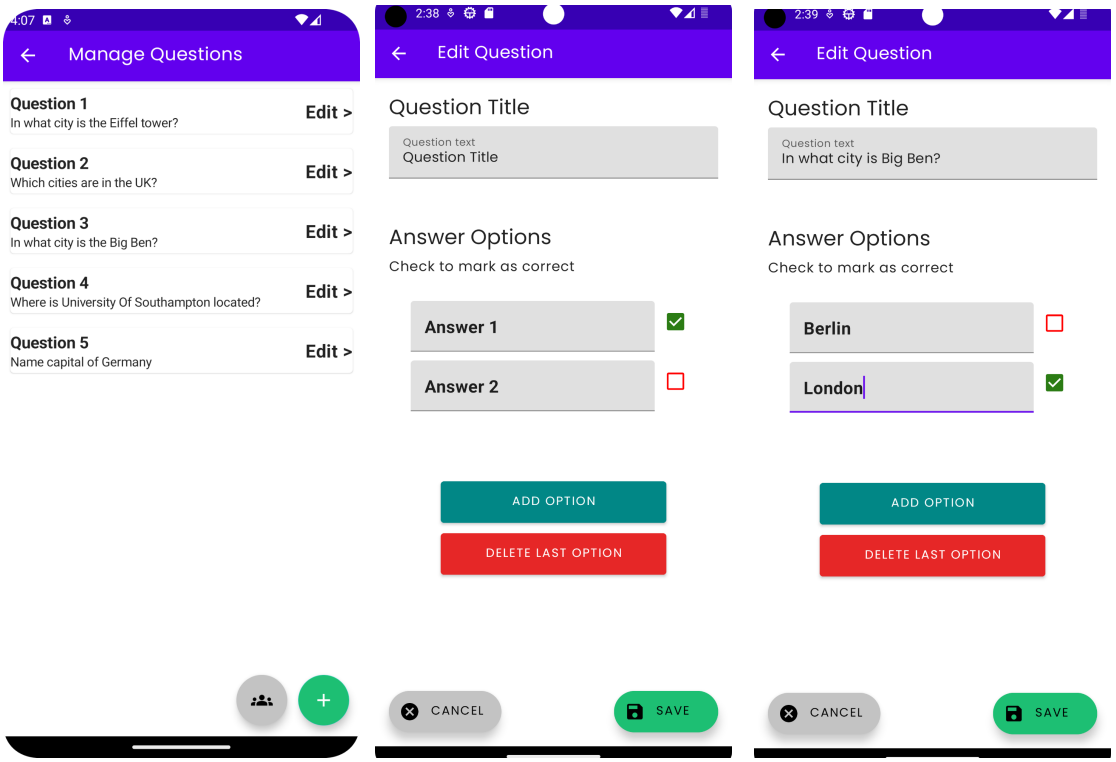

Figure 16: Manage Questions

The fragment responsible for editing or adding a question is implemented as a mix of Compose and XML file layouts. For example, an answer option is a Composable that involves input consisting of a textfield and a checkmark next to it shown in the code of the figure below. When working with input components like TextField and Checkbox as in our case, we have to use state objects to manage the state of these components. This is done in the checkboxState and textFieldState variables for their respective Checkbox and TextField compose components. We are following the example provided in slide 35 of week 6 of lecture notes [6].

```
LeonidGoldberg +1
@Composable
private fun AnswerOption(mapID: Int) {
    Row(modifier = Modifier.padding(5.dp)) { this: RowScope
        val checkboxState = remember {
            mutableStateOf(answerOptionsValues[mapID]!!.second)
        }
        val textFieldState = remember {
            mutableStateOf(answerOptionsValues[mapID]!!.first)
        }

        TextField(
            modifier = Modifier.weight(1f),
            value = textFieldState.value,
            onValueChange = { it: String
                textFieldState.value = it
                answerOptionsValues[mapID] = Pair(it, checkboxState.value)
            })
        Checkbox(
            checked = checkboxState.value, onCheckedChange = { it: Boolean
                checkboxState.value = it
                answerOptionsValues[mapID] = Pair(textFieldState.value, it)
            }, colors = CheckboxDefaults.colors(
                checkedColor = Color( red: 43,  green: 124,  blue: 20,  alpha: 255),
                uncheckedColor = Color.Red
            )
        )
    }
}
```

Figure 17: Using Jetpack Compose for displaying Answer Options
Filepath: app/src/main/java/com/example/educationsupport/ui/learneractivities/EditQuestionFragment.kt

Activities and Results

When the learner clicks on an activity they see an activity introduction screen with its description and the number of questions. They can either start the activity or go back to the list of activities. They are then presented with the activity's questions. They must select at least one answer to be able to submit the question. Learners can navigate between questions using the arrow buttons. Once all questions are answered, they can submit the activity. Then they see their results on the results screen. If a question was answered incorrectly, the app shows which answers the user selected incorrectly as well as the correct answers that the user didn't select. The screens are shown in the figure below.

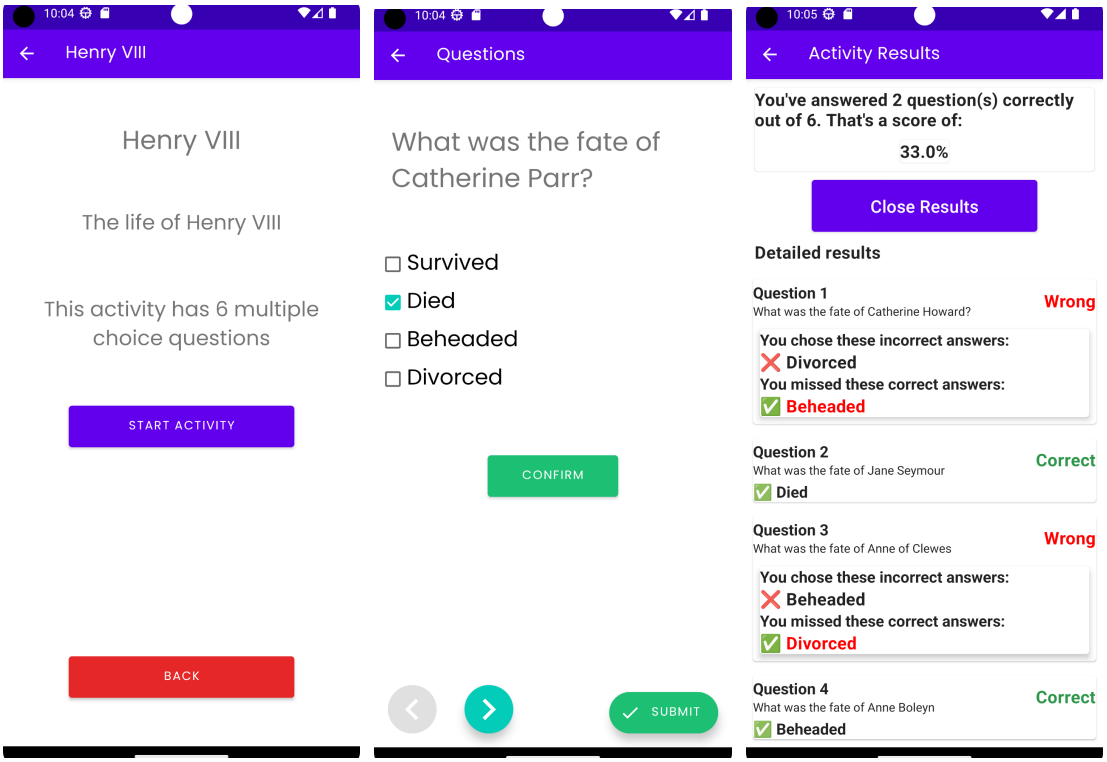

Figure 18: Learner taking the quiz and viewing the results

We show the results of the activity to the learner using Jetpack compose. The composable function in Figure 19 on the next page is responsible for displaying a single result item to the user:

```
Leonid Goldberg +2
@Composable
private fun ResultItem(
    questionNumber: Int,
    question: Question,
    correct: Boolean
) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(8.dp),
    ) {
        Column { this: ColumnScope
            Row(
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.SpaceBetween,
                modifier = Modifier
                    .fillMaxWidth()
                    .height(50.dp)
            ) { this: RowScope
                Column { this: ColumnScope
                    Text(
                        text = "Question $questionNumber",
                        style = MaterialTheme.typography.subtitle1
                    )
                    Text(
                        text = question.questionText,
                        style = MaterialTheme.typography.subtitle2
                    )
                }
                Column { this: ColumnScope
                    Text(
                        text = if (correct) {
                            "Correct"
                        } else {
                            "Wrong"
                        },
                        color = if (correct) {
                            Color( red: 45,  green: 150,  blue: 73,  alpha: 255)
                        } else {
                            Color.Red
                        }
                    )
                }
            }
            Row { this: RowScope
                if (!correct) {
                    detailedResultInfo(indexOfQuestion = questionNumber - 1)
                }
            }
        }
    }
```

Figure 19: Using Jetpack Compose to show the results
Filepath: app/src/main/java/com/example/educationsupport/ui/learneractivities/ActivityResultsFragment.kt

Once a learner has completed an activity, the educator who created the course can then see the results of that test.

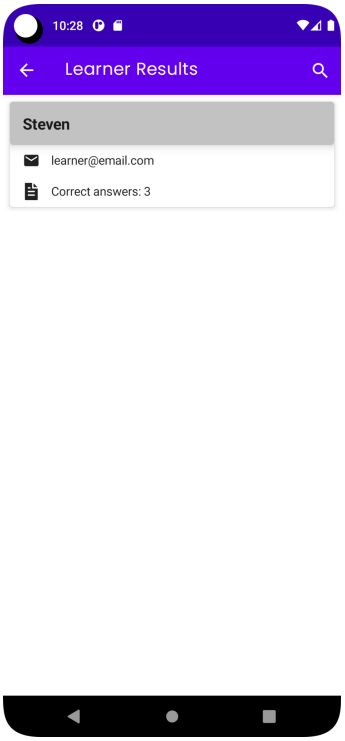

Figure 21: Course Results Page

Account Page

The user can customise their profile by editing the name, bio and profile image (F10). To do so they can navigate to their Profile Page from the menu in the top-right corner.

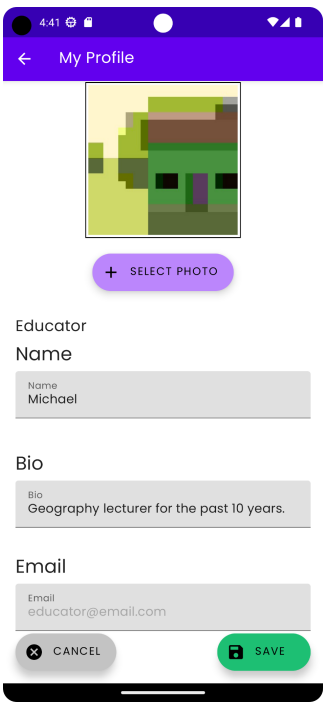

Figure 22: My Profile Page

Backend

For the backend and the database we have used Firebase [2].

Authentication

To authenticate users, we have taken advantage of the Firebase Authentication functionality (F11). Using this tool means that we do not have to deal with the specifics and security of authentication methods. In our case, we have implemented a simple sign up and log in using the user's email and password. To do this, we have relied on the relevant section of the Firebase documentation [3].

Firebase Database

We have decided to use the NoSQL Firebase Realtime Database [4] (F11) as some of our team members were familiar with working on it and was faster to implement. It allowed us to pull data from the database with little added latency. This is especially useful for mobile devices that tend to have unreliable connections. We didn’t need to use any advanced features for our application like those that the Firestore Database provides. To manipulate the data in the database, we have created a Firebase class that contains references to all the collections, and the methods we need for adding items to the database and working with them. It also contained template code that was used to more easily implement our functions. The code snippet below, shows an example of how we would use the Firebase API to add a question to the database, and then link it to an activity.

```kotlin
/**
 * Adds a new question to an existing activity.
 *
 * @param activity        the activity object to add the question to
 * @param activityID    the id of the activity to add the question to
 * @param question    the question object to be added to activity
 */
fun addQuestionTo(activity: Activity, activityID: String, question: Question): String? {
    // Generating a new questionID
    val questionID = questionsTable.push().key
    if (questionID != null) {
        // Adding new question to the database
        questionsTable.child(questionID).setValue(question)
            .addOnSuccessListener { it: Void!
                Log.d(
                    tag: "Firebase Realtime Database",
                    msg: "Question $questionID was added!"
                )
            }.addOnFailureListener { it: Exception
                Log.e(
                    tag: "Firebase Realtime Database",
                    msg: "Unable to add question $questionID!", it
                )
            }

        // linking this question to activity
        activity.questionIDs[questionID] = true
        editActivity(activityID, activity)
        return questionID
    } else {
        Log.e( tag: "Firebase Realtime Database",  msg: "Unable to generate a new questionID!")
        return null
    }
}
```

Figure 22:
Filepath: app/src/main/java/com/example/educationsupport/db/Firebase.kt

Tools and Libraries Used

Tool	Justification	Evaluation
Kotlin 1.8.21	The main programming language used for the development of the application.	The syntax was easy to pick up and more concise than Java resulting in improved code readability.
JDK 17(or newer)	We needed to use JDK 17 or newer compatible versions to compile the app	We specifically used JDK 17 to compile our app, but any version compatible with Kotlin 1.8.21 should be able to run it.
Gradle	Gradle was the building tool we used for this project.	Gradle is the default implementation used in Android Studio. It allowed us to easily setup and add libraries while also keeping them up to date.
Jetpack Compose	Jetpack Compose elements were used for displaying list elements from code and allowed us to more easily link data to code elements	Creating UI elements directly from Kotlin made for a more cohesive development experience.
Firebase Realtime Database	Firebase RD was used as our cloud database for storing our objects and requesting the data back to the app.	Using Firebase services was very convenient as they have great integration with each other and are all accessible through the same console and SDK. These tools saved the team a lot of time since we didn't have to manually handle authentication and storage.
Firebase Authentication	We used FA for making secure user accounts registrations and logins.	
Firebase Storage	Firebase Storage was used for storing user profile pictures on the cloud.	
Bumpteck's Glide	A Github library, used for loading and displaying pictures from URL	These libraries made the personalisation of a profile much easier and convenient.
Dhaval2404's Image Picker	A Github library, used for picking pictures from the phone's library or taking a camera picture.	

Testing

The testing was done manually against the specification requirements throughout the development process as well as at the end. The things considered in the testing were:

- **Functionality**: All the implemented features were tested from the perspective of the different stakeholders (educators and learners) and worked as expected. Android Studio's Logcat was used to log firebase responses and for debugging the functionality of the app.

- **Compatibility** through Emulator System Testing: The application was tested on a variety of devices from different platforms including: Pixel, Samsung and Nexus. Furthermore within the layout view, the Design view allowed us to test different screen sizes and resolutions from the provided preset devices.

- **Responsiveness**: The application was tested on mobile and tablet screen sizes.

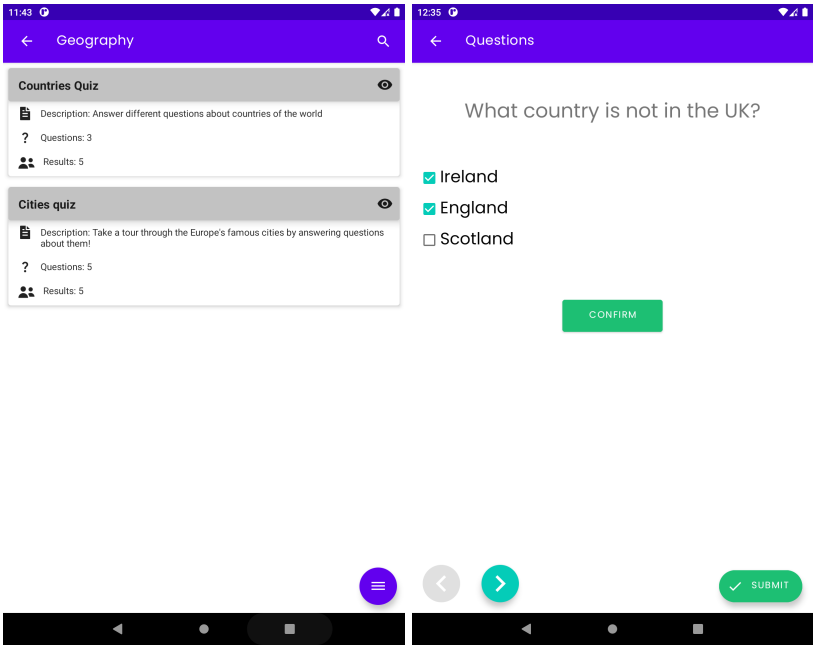

Figure 21: Nexus 7 Tablet Emulated device screen scaling example

- **Accessibility**: The design was chosen with accessibility best practices in mind to ensure enough contrast between the theme colours.

References

[1] Atlassian (2019). *Jira Cloud*. [online] Atlassian. Available at: https://www.atlassian.com/software/jira.

[2] Google (n.d.). *Firebase*. [online] Firebase. Available at: https://firebase.google.com/.

[3] Google (n.d.). *Get Started with Firebase Authentication on Android*. [online] Firebase. Available at: https://firebase.google.com/docs/auth/android/start?hl=en&authuser=1 [Accessed 6 Apr. 2023].

[4] Firebase. (n.d.). *Connect your App to Firebase | Firebase Realtime Database*. [online] Available at: https://firebase.google.com/docs/database/android/start?hl=en&authuser=1 [Accessed 6 May 2023].

[5] Figma (n.d). *Figma: The Collaborative Interface Design Tool* [online] Available at: https://www.figma.com [Accessed 8 May 2023].

[6] COMP6239-Lecture13-Jetpack-Compose-2023.pdf Available at: https://secure.ecs.soton.ac.uk/noteswiki/images/COMP6239-Lecture13-Jetpack-Compose-2023.pdf