

University of Greenwich

COMP1786 Coursework

Instructor: Mr. Hoang Nhu Vinh

001428450

11-28-2024

Table of Contents

1. BRIEF STATEMENT OF FEATURES YOU HAVE IMPLEMENTED (2%)	2
2. REFLECTION (4%)	3
3. EVALUATION (10%):	4
4. DESIGN (2%):.....	6
1. Android	7
2. React Native	24
5. CODE (2%)	30
1. Android	30
2. React Native	44
Table of Figures	49
Reference List	50

Your name	Bui Quang Minh	Your Student ID	001428450
------------------	-----------------------	------------------------	------------------

1. BRIEF STATEMENT OF FEATURES YOU HAVE IMPLEMENTED (2%)

Feature	Status	Your Comments
Functionality A	Fully completed <input checked="" type="checkbox"/> Partially completed <input type="checkbox"/> Having bugs/Not working <input type="checkbox"/> Not implemented <input type="checkbox"/>	The function is fully implemented and fields are validated without any errors. To improve user experience, I used and customized elements such as a spinner, time picker, and radio buttons. I also applied blue as the main color and a custom theme for pickers and buttons.
Functionality B	Fully completed <input checked="" type="checkbox"/> Partially completed <input type="checkbox"/> Having bugs/Not working <input type="checkbox"/> Not implemented <input type="checkbox"/>	I implemented a DAO pattern to separate the persistence and business layers. This helps to effectively modify and update the local database for courses and classes. All functions run effectively. Data stored in the SQLite database also works well.
Functionality C	Fully completed <input checked="" type="checkbox"/> Partially completed <input type="checkbox"/> Having bugs/Not working <input type="checkbox"/> Not implemented <input type="checkbox"/>	The class instance is implemented and validated effectively. The class's date is validated strictly to match the day of the week that belongs to its course which helps to improve user experience and the class instance's data also is stored in the SQLite database effectively.
Functionality D	Fully completed <input checked="" type="checkbox"/> Partially completed <input type="checkbox"/> Having bugs/Not working <input type="checkbox"/> Not implemented <input type="checkbox"/>	I implemented the search function fully such as filtering by name, comment, and by date of week. The days of the week can be scrolled horizontally with active days that help to enhance user experience.
Functionality E	Fully completed <input checked="" type="checkbox"/> Partially completed <input type="checkbox"/> Having bugs/Not working <input type="checkbox"/> Not implemented <input type="checkbox"/>	I applied an advanced feature which is a work manager to synchronize data periodically or we can synchronize it immediately by clicking a button. The data can be tracked from local to the cloud by status including add, update, and delete and the connection also is checked before uploading to the cloud.
Functionality F	Fully completed <input checked="" type="checkbox"/> Partially completed <input type="checkbox"/> Having bugs/Not working <input type="checkbox"/> Not implemented <input type="checkbox"/>	I used glide library with some custom animation to create a beautiful carousel to display marketing banners that make the application more attractive.
Functionality G	Fully completed <input checked="" type="checkbox"/> Partially completed <input type="checkbox"/> Having bugs/Not working <input type="checkbox"/> Not implemented <input type="checkbox"/>	I set up a structural code and picker to create elegant interfaces for fetching and filtering classes by search term and date of week. Also, clicking on each item can show the class instance data with class details data.
Functionality H	Fully completed <input checked="" type="checkbox"/> Partially completed <input type="checkbox"/>	I implemented tab navigation to create more screens for cart and profile screens to help users add classes to the

	Having bugs/Not working <input type="checkbox"/> Not implemented <input type="checkbox"/>	cart and book classes so that users can view classes in the cart and book classes that user already booked before. Data about carts and orders also uploaded and stored in Firebase
--	--	--

Link to recorded video (if you record your application before submitting the report) The recorded video will demonstrate the implemented product. It is roughly 10 minutes. https://drive.google.com/file/d/1e8sLK_OddjpYP_35bFH9FXlGvCFjETEM/view?usp=drive_link
--

2. REFLECTION (4%)

Developing the Universal Yoga applications for Android and React Native was an exciting journey. Current technologies, user-oriented design ideas, and efficient problem-solving had to be mixed to create practical and understandable apps.

The development process depended critically on a structured way. From the Android side, I enhanced maintainability by using SQLite for local storage and the DAO design to split business logic from persistence. Managing regular data synchronizing chores proved very helpful with WorkManager. TypeScript was quite important for React Native in guaranteeing type safety and lowering flaws. Its modular design let for scalable and reusable components, hence facilitating cross-platform development.

Several features including synchronizing systems to control add, update, and delete activities between local storage and the cloud were put in use effectively. While easy tab navigation permitted seamless movement across app displays, Glide helped to create a visually interesting carousel enhanced user experience. Customizing themes and UI elements improved consistency, hence producing an aesthetically beautiful and user-friendly experience.

Even with these successes, there remained areas needing improvement. React Native made responsiveness on larger devices such as tablets challenging. Even though I used flex-based layouts, integrating percentage-based measures and media searches will provide better experience on bigger screens. Sometimes synchronizing React Native and Android apps generated inconsistent data since edge situations not entirely supported. Solving these issues would depend on better error control and building a single synchronization mechanism. Firebase Crashlytics can support dependability and debugging capacity enhancement.

Security also presented development prospects. SQLite data was kept without encryption, therefore exposing possible hazards for private data. Reducing these concerns would be adding SQLCipher for encrypted local storage and guaranteeing TLS safe data delivery. Moreover, using OAuth 2.0 will improve security of authentication.

All things considered, this project show how well TypeScript, React Native, and Android were merged to produce scalable, feature-rich apps. While significant benchmarks were achieved, areas including timeliness, data synchronization, and security call for greater development even if overall standards were met. This understanding underscored in software development the requirement of structure, flexibility, and user-first thinking.

3. EVALUATION (10%):

The Universal Yoga apps were designed to simplify yoga class management for admins and provide an intuitive booking system for customers. This evaluation will cover many aspects, including usability, security, adaptability, performance, and maintainability

Human-Computer Interaction (HCI)

To allow users to navigate and interact with the applications easily and make them user-friendly and intuitive. The usability of the apps was evaluated using Nielsen's 10 Usability Heuristics (Nielsen, 1994). Features such as dropdown menus, spinners for selecting days, time pickers, and radio buttons align with the heuristics of Consistency and Standards and Match Between System and the Real World. Horizontal scrolling for day selection enhances efficiency, meeting the principle of Flexibility and Efficiency of Use. For day choosing, horizontal scrolling improves efficiency and satisfies the flexibility and efficiency of use principle. Building these is not too tough based on the subject contents and my classroom instructor. Horizontal scrolling for day selection also improves efficiency, therefore fulfilling the idea of flexibility and efficiency of use.

There are many ways to create intuitive navigation, such as a drawer and options menu, but after trying some current famous applications, Tab-based navigation was selected for its familiarity to users, reducing cognitive load and learning time (Shneiderman's 8 Golden Rules of Interface Design, 1998), it reduces the need for guidance and learning time for users. This helps users to switch between screens easily. To further improve usability, applying WCAG 2.1 Guidelines for accessibility, such as scalable fonts and contrast checks, would ensure inclusivity for users with disabilities.

Security and Data Protection

Both Android and React Native employ basic security measures, such as validating every input to safeguard data integrity and prevent application crashes, as well as confirming required fields during data entry. The Android application synchronizes data from local storage to the cloud, ensuring consistency; however, substantial enhancements are necessary. The security protocols were evaluated in accordance with the OWASP Mobile Security Guidelines (OWASP, 2023).

Without user authentication, one runs the danger of unwanted access. Following OWASP best standards means including bcrypt for password encryption and OAuth 2.0 for safe login.

SQLite data storage is not encrypted, hence sensitive data runs the danger. Data at rest and in transit would be safeguarded by encrypted storage using SQLCipher and TLS 1.2 or above guaranteeing safe transfer.

Firebase Crashlytics could be included to record security concerns and running time mistakes. Technical information should not be disclosed in user-facing error messages to stop exploitation.

Adaptability Across Devices

Device adaptability was assessed using Google's Material Design Guidelines (Google, 2023) and Apple's Human Interface Guidelines (Apple, 2023). The Android app uses multiple layouts, including ConstraintLayout, which enables scalability across screen sizes. However, inconsistencies in larger devices like tablets could be addressed by implementing responsive design principles, including percent-based layouts and breakpoints.

Cross-platform support is guaranteed by React Native, but small styling differences (like button alignment) were seen because of differences in how each platform renders. These problems can be

fixed by using tools like Styled-Components and react-native-responsive-screen, which use dynamic sizing and media queries. Using tools like BrowserStack to try on a lot of different devices with different screen sizes and aspect ratios would ensure the best performance.

Deployment and Live Use

Getting the apps ready for real-world use requires fixing a few key issues to make them more reliable and user-friendly.

Currently, the apps do not handle all possible errors, like failed data syncing and consistency between carts and orders. Adding tools like Firebase Crashlytics can help find and fix issues by tracking where things go wrong. Simple, clear error messages and retry options for network problems would also make the apps easier to use.

Fields including Created By, Created Date, Updated By, and Updated Date can be added to models for both SQLite and Firebase to increase data accuracy and traceability, therefore satisfying ISO 25010:2011 criteria for maintainability. These disciplines will enable tracking of changes, point up problems during synchronizing, and offer handy logs for crash troubleshooting. This guarantees more data control and facilitates debugging.

The applications must to be tested under several network environments and on several distinct devices and screen sizes. This covers low-memory conditions, odd user inputs, slow internet testing, and loading indicator addition to guarantee they function everywhere.

Maintainability and Scalability

Maintainability was evaluated using the ISO 25010:2011 Software Quality Model, focusing on modularity and testability.

When starting to build the project from base code, maintainability is always the main criterias that should be the focus, so the code that I write is structural and clean. In the Android app, the DAO pattern is applied to separate business logic from database operation. This makes the database interactive tasks easier to manage and modify and reduces the risk of errors during the development environment. React Native utilizes TypeScript, which is a strict type checking, so it improves code quality and minimizes bugs.

The database architecture has to be designed to properly handle large amounts of data. As supported by the ideas of database normalisation, indexing often requested fields, like instructor names and dates, in SQLite may improve efficiency (Elmasri & Navathe, 2015).

Including Firebase Crashlytics not only detects runtime failures but also generates thorough logs that help to find and fix problems. Recording important events like data synchronization and database updates could help to monitor the performance of the program under growth.

4. DESIGN (2%):

At first, I spent time visualizing my ideas of interfaces on paper to make them clearer. I tried many ways to find out which one is the most suitable for the applications.

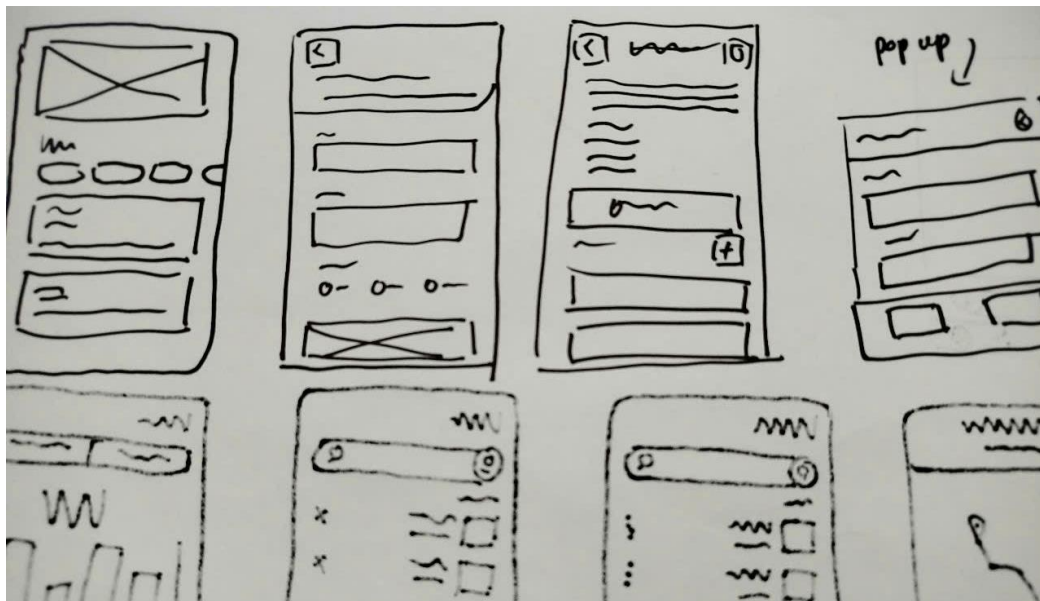


Figure 1. Design Sketching

I also tried some designs with different colors to decide which color scheme and design were the best for the application on Figma. Then, I chose the blue one, which made me feel more fresh, clean, and intuitive. But in the future, when I deploy it to the internet, I need to survey to get feedback from real users, and it would be more reliable and concise to decide which one is better.

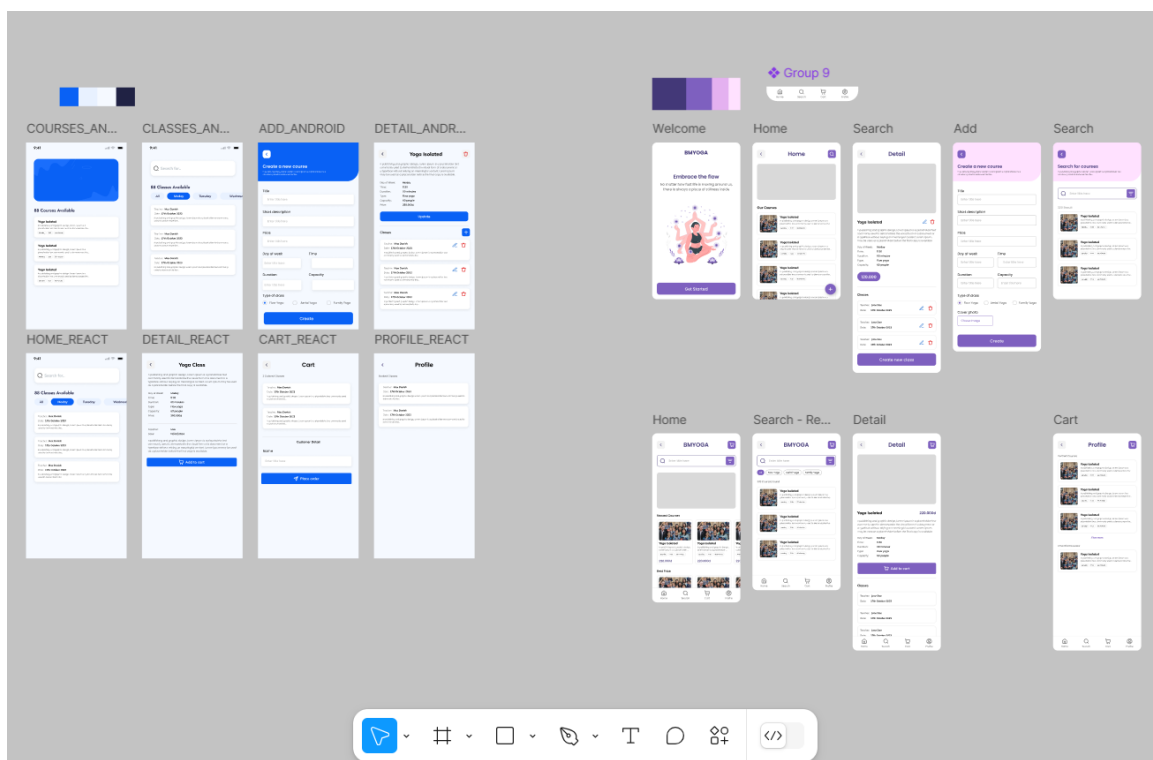


Figure 2. Figma Design

1. Android

A lot of custom for UI and style to create interfaces that match 90% compared to the design such as animation, color scheme, theme, menu and drawable.

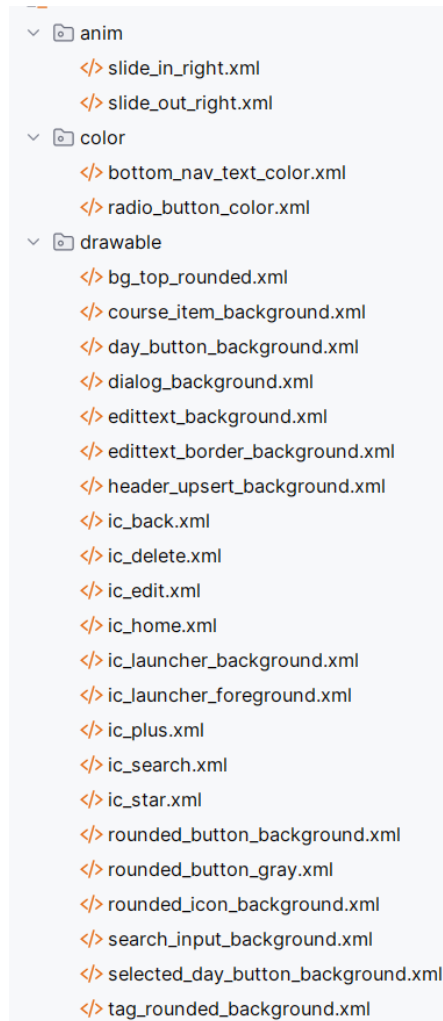


Figure 3. res Folder (1)

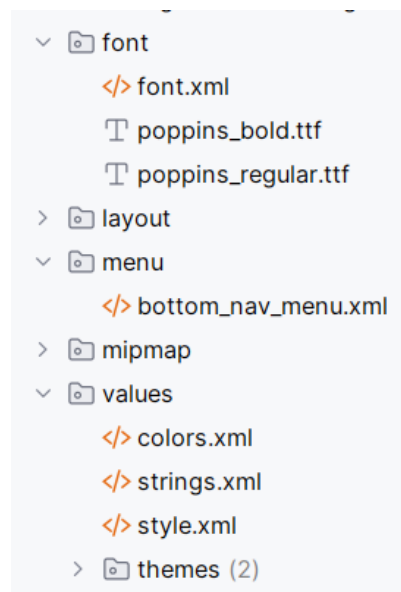


Figure 4. res Folder (2)

From now on, I will explain the flow of the applications, focusing on the screens with which users see and interact.

In the **Courses** tab, there is a carousel, which is a slideshow showing banners with animation. Below that would be a list of course cards that contain featured information, including name, description, day of week, time, and duration.

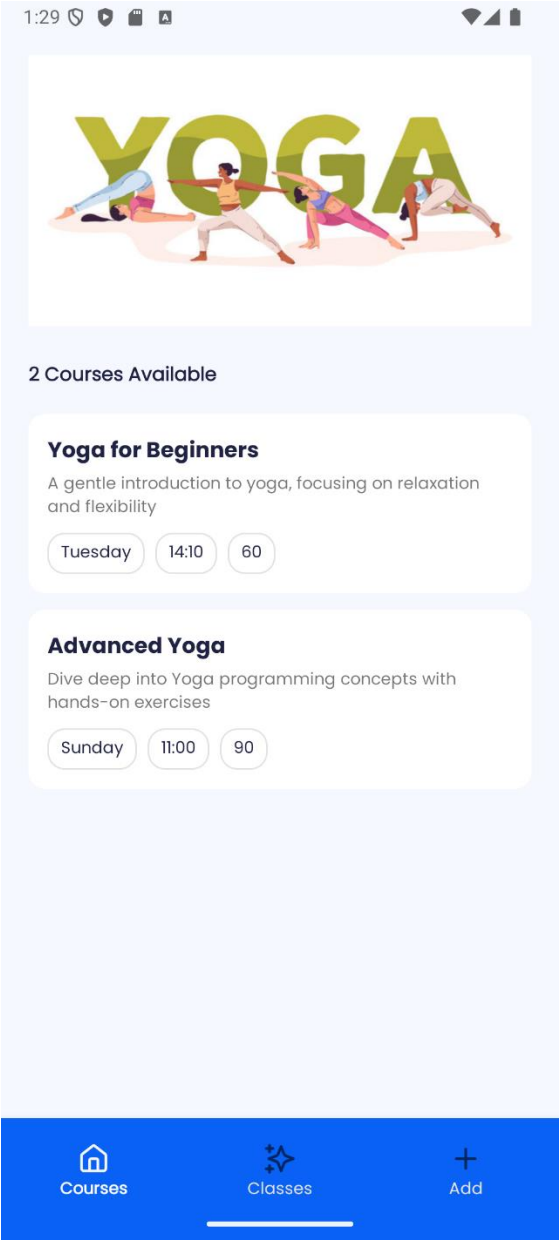


Figure 5. Courses Screen

When clicking on a **course item** in the **Courses Screen**, it will direct to the **Course Detail screen**, I also designed a very intuitive one with a clear back, delete, add, and update buttons. All course detail is shown first, and then a list of classes is shown to help users know that classes belong to the course.

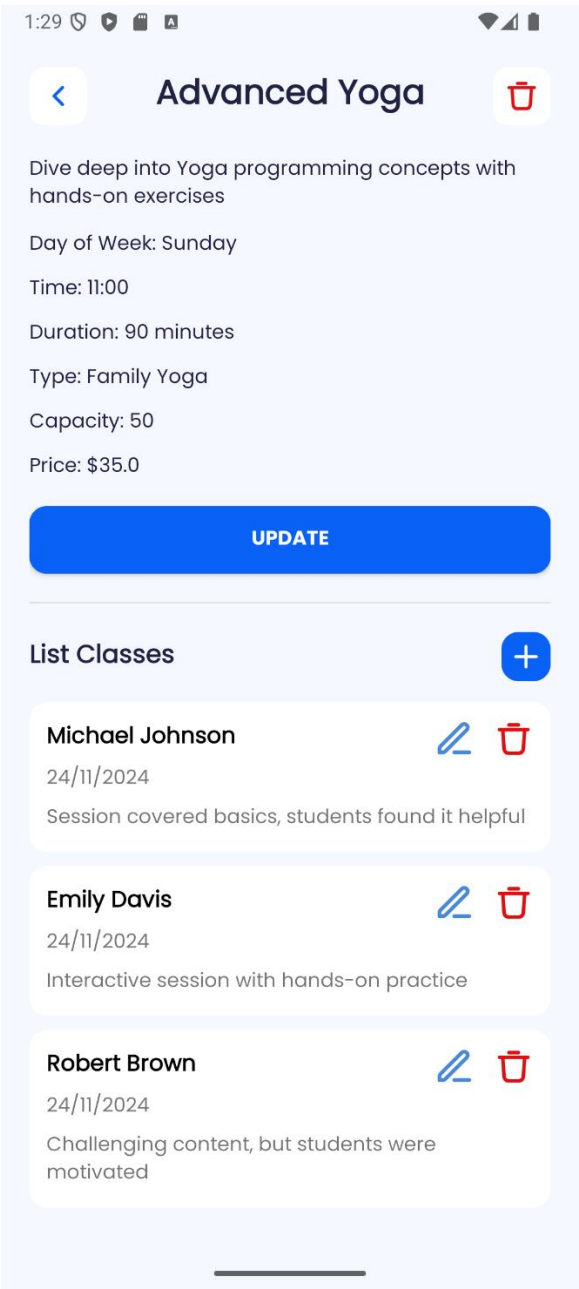


Figure 6. Course Detail Screen

The **Classes** tab contains a search input, horizontal scrolling for a day of the week which is also easy and intuitive to use, and a list of the class cards. Each class will show the teacher's name, date, and comment.

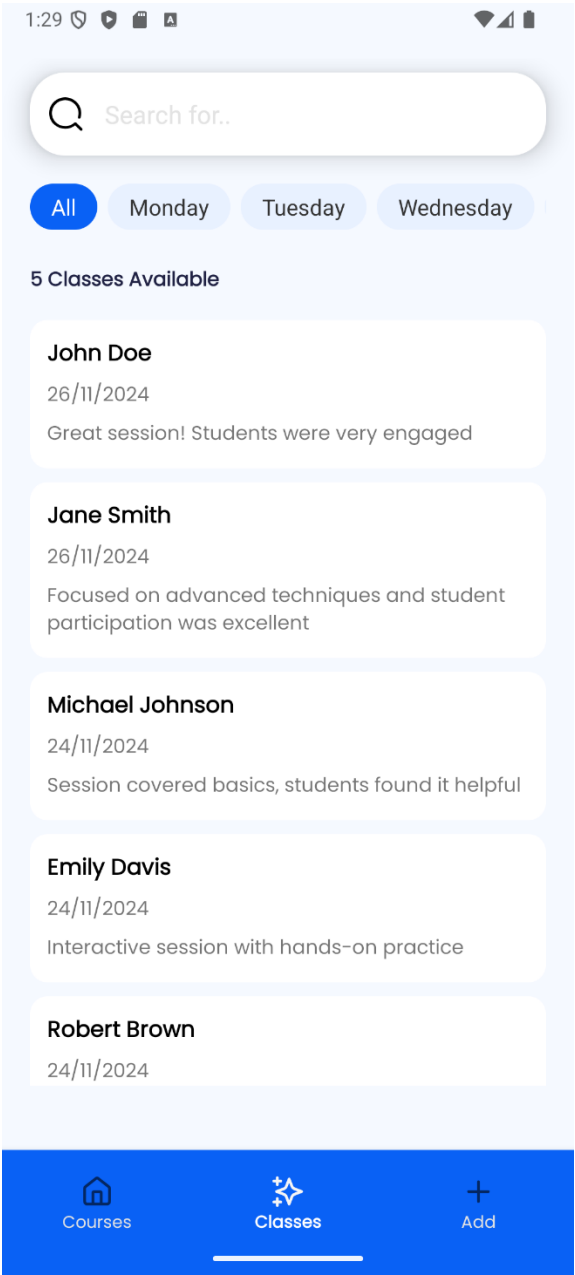


Figure 7. Classes Screen

When entering the search term and selecting the day, the classes will be filtered to match. The search term will be searched in not only the teacher's name but also in comments to not miss any information.

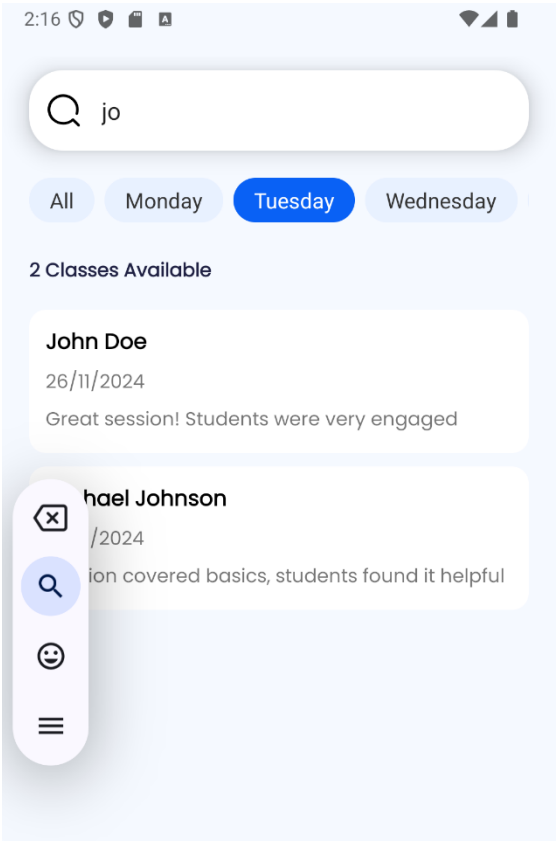


Figure 8. Filtered Classes

In the **Adding Course Screen**, I designed this very carefully with a beautiful rounded header. Using blue as the main color and customizing the time picker and radio buttons to a blue color to make consistency for the whole screen.

1:23

Create a new course

In publishing and graphic design, lorem ipsum is a placeholder text...

Name

Enter name

Short Description

Enter description

Price

Enter price

Day of week **Time**

Monday Select time

Duration **Capacity**

Enter duration Enter price

Type of class

☒ Flow Yoga ☐ Aerial Yoga ☐ Family Yoga

ADD COURSE

Courses Classes Add

Figure 9. Add Screen

Enter description

Price

Enter

Set time

2 38

hour minute

Day of week

Monday

Duration

Enter duration Enter price

Figure 10. Time Picker

When the user clicks to add a course without filling in full information, the error will be displayed on each field, and a Toast will be shown at the same time. However, the error will be shown from top to bottom fields to ensure that it does not check full fields at a time.

2:20

Create a new course

In publishing and graphic design, lorem ipsum is a placeholder text...

Name

Enter name

Short Description

Enter description

Price

Enter price

Day of week

Monday

Time

Select time

Duration

Enter duration

Capacity

Enter price

Type of class

☒ Flow Yoga

☐ Aerial Yoga

☐ Family Yoga

Please fill in all fields

Courses





Classes




Add

Figure 11. Errors without Fulfill Information

When filling in the information and clicking the add button will show a success toast and all fields will be reset to the first state.

2:24





Create a new course

In publishing and graphic design, lorem ipsum is a placeholder text...

Name

Short Description

Price

Day of week

Time

Duration

Capacity


Type of class


☐ Flow Yoga


☐ Aerial Yoga

☒ Family Yoga

ADD COURSE

 Courses

 Course created successfully

 Add

Classes

Figure 12. Adding Course Successfully

When clicking on a big update button on the **Course Detail Screen**, it navigates to the Update course screen, and all the fields, such as input, radio, and spinner, are set values.

1:29

<

Update course

In publishing and graphic design, lorem ipsum is a placeholder text...

Yoga for Beginners

Short Description

A gentle introduction to yoga, focusing on relaxation and flexibility

Price

100.0

Day of week	Time
Tuesday	14:10

Duration	Capacity
60	100

Type of class

☐ Flow Yoga ☒ Aerial Yoga ☐ Family Yoga

UPDATE COURSE

Figure 13. Update Course Screen

If clicking the update course button without fulfilling information, the error message also be shown on the input to make sure the user enters the information fully.

The screenshot shows a mobile application interface for updating a course. At the top, there is a blue header with a back arrow icon and the title 'Update course'. Below the header, there is a text area for the course title, currently containing 'Yoga for Beginners'. This is followed by a 'Short Description' section with a text area containing 'A gentle introduction to yoga, focusing on relaxation and flexibility'. The 'Price' section has a text input field with the placeholder 'Enter price', which is highlighted with a red border and a red exclamation mark icon. A black tooltip with the text 'Price is required' points to the red icon. Below the price field, there are two columns of form fields: 'Day of week' with a dropdown menu showing 'Tuesday', and 'Time' with a dropdown menu showing '14:10'. Further down, there are 'Duration' and 'Capacity' fields, both with dropdown menus showing '60' and '100' respectively. The 'Type of class' section has three radio button options: 'Flow Yoga', 'Aerial Yoga' (which is selected), and 'Family Yoga'. At the bottom of the form is a large blue button labeled 'UPDATE COURSE'. The entire form is set against a light blue background.

Figure 14. Error Message when Updating

When updating the course successfully, it will show a toast message to let the user know that the course is updated and navigate back to the details with the latest information.

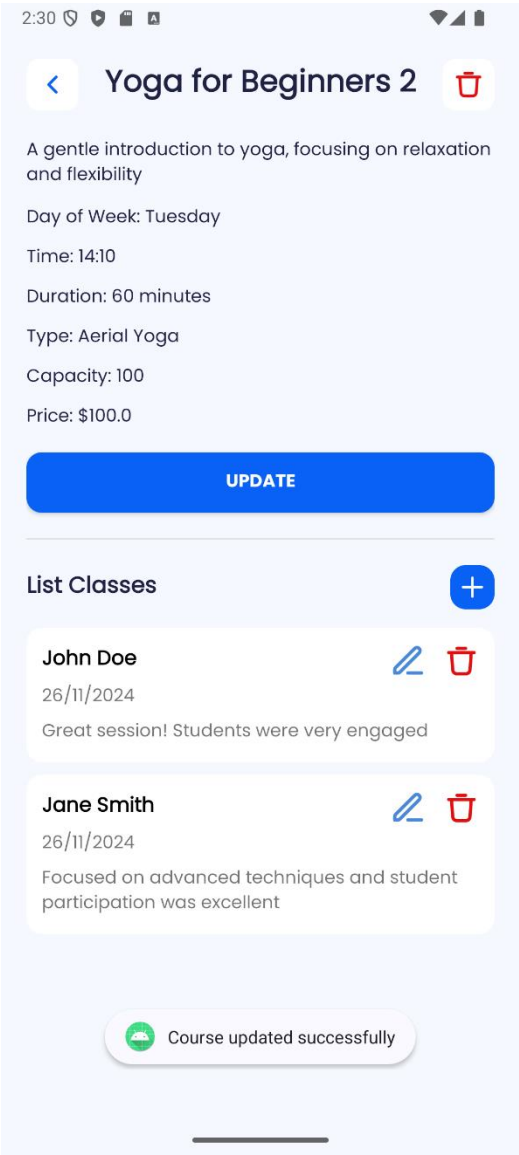


Figure 15. Update Course Successfully

When you click the trash button on the top right corner, a confirm dialog will pop up to delete the course. If the user clicks the cancel option, the dialog will be hidden and nothing happen.

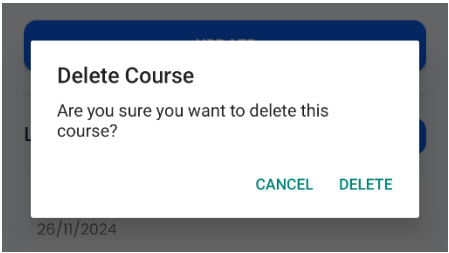


Figure 16. Delete Confirm Dialog for Course

When clicking on the delete option, the course will be deleted and it will show a toast with messages and navigate back to the Courses screen without the deleted course.

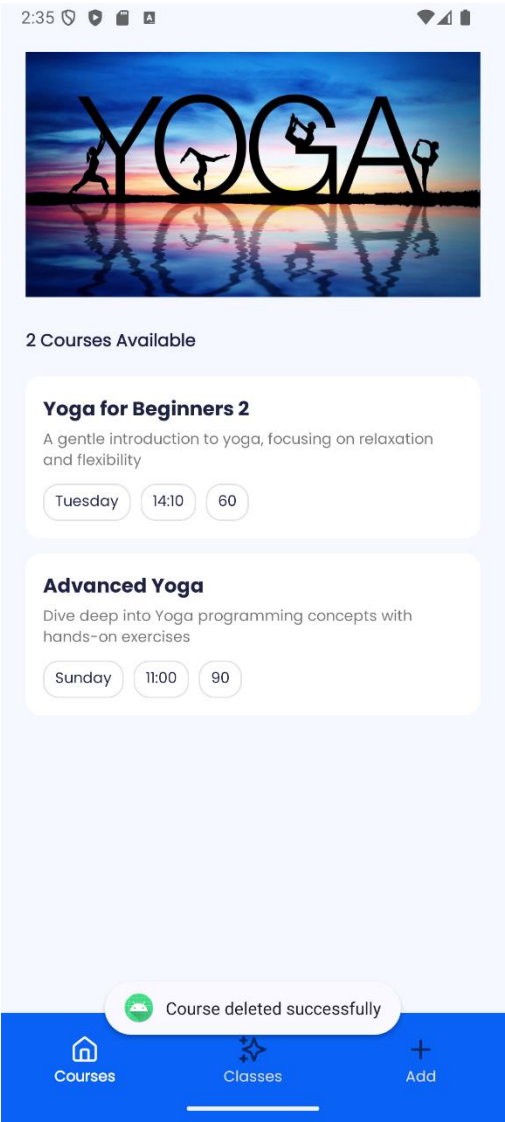


Figure 17. Delete Course Successfully

When clicking the blue plus button, which is the same line as “Classes,” will pop up a window that contains fields including the Teacher's name, Date of week, and Comment. and two buttons: one is to hide the dialog, and one is to add the current class.

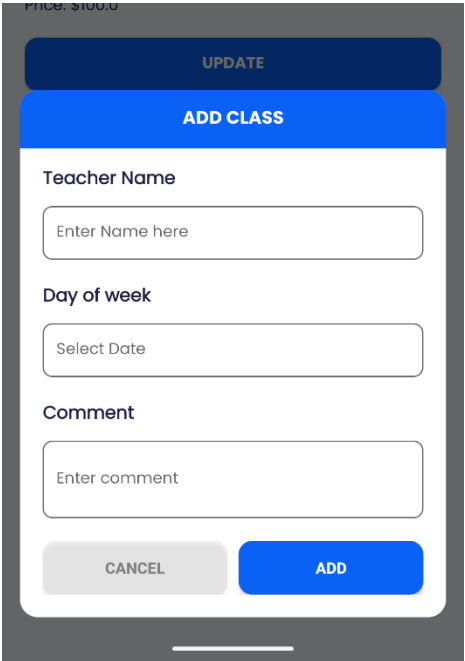


Figure 18. Add Class Dialog

If the user clicks the add button without fulfilling information error messages will show on each input and the comment is optional. If the user selects the date but does not match the day of the week of course, the toast message will be shown to remind the user to select the correct date.

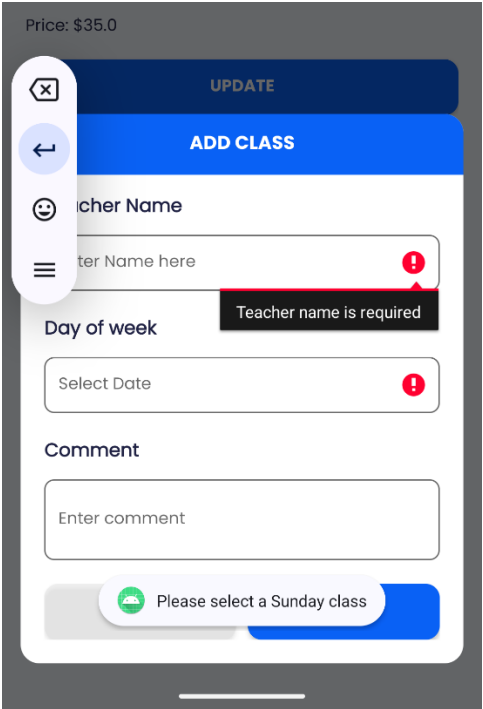


Figure 19. Error Messages when Adding Class

Clicking on the date of week input will pop up a date picker, which I custom color following the main theme.

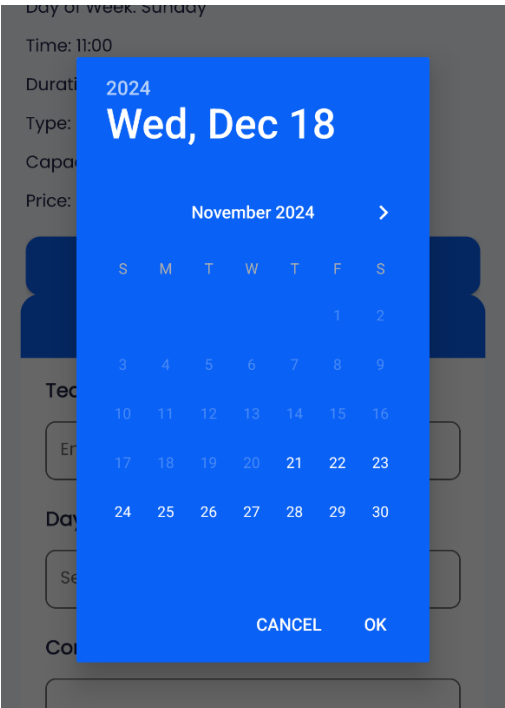


Figure 20. Date Picker

When the user fills in the required information and clicks the add button. A toast will be shown to inform that the class is added successfully and it navigates back to the Course Detail screen with a new class.

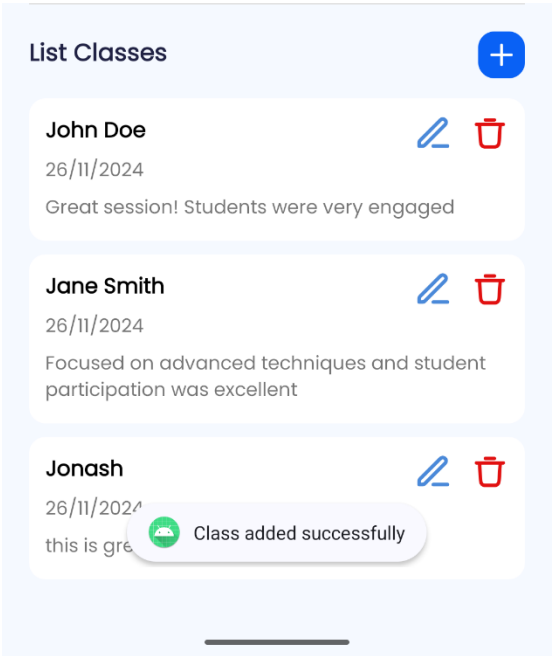


Figure 21. Add Class Successfully

When you click the pen button on each class, the same dialog for adding a class will pop up, but now, it is filled with the class's details. All validation steps are the same with adding.

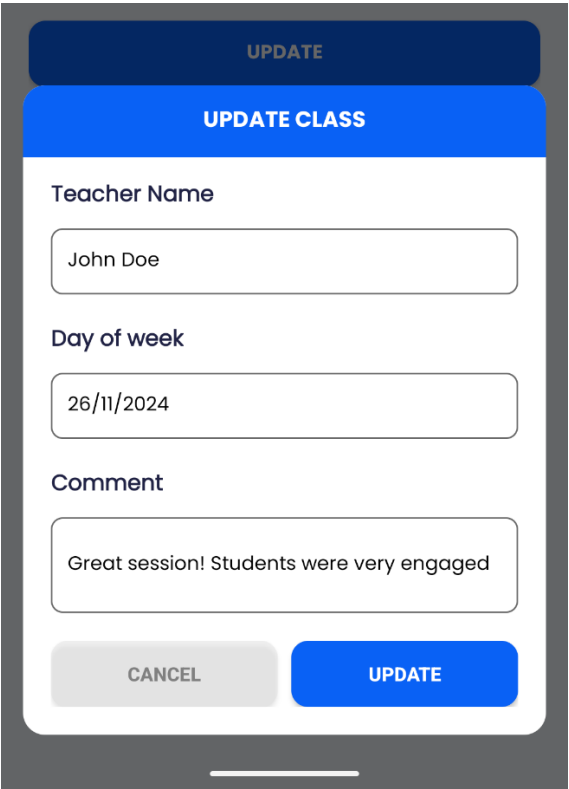
A mobile app dialog titled "UPDATE CLASS" with a dark blue header bar containing the word "UPDATE" in white. The dialog has a white background with rounded corners. It contains three text input fields: "Teacher Name" with the value "John Doe", "Day of week" with the value "26/11/2024", and "Comment" with the value "Great session! Students were very engaged". At the bottom, there are two buttons: a grey "CANCEL" button and a blue "UPDATE" button.

Figure 22. Update Class Dialog

When filling in the required information with new data and clicking on the update button will show a toast message and navigate back to the detail screen with the updated class.

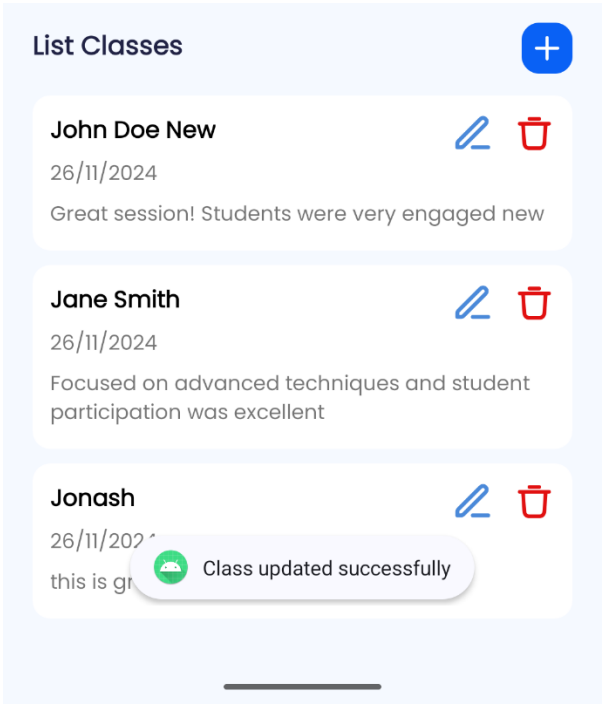
A mobile app screen titled "List Classes" with a light blue background. At the top right is a blue circular button with a white plus sign. Below the title is a list of three class items. Each item has a title, a date, and a comment, followed by a blue pen icon and a red trash icon. The first item is "John Doe New" with date "26/11/2024" and comment "Great session! Students were very engaged new". The second item is "Jane Smith" with date "26/11/2024" and comment "Focused on advanced techniques and student participation was excellent". The third item is "Jonash" with date "26/11/2024" and comment "this is gr". A green toast message with a white checkmark icon and the text "Class updated successfully" is overlaid on the bottom of the list.

Figure 23. Update Class Successfully

When clicking to trash button on each class, a confirm dialog will show to allow the user to delete the class from the course.

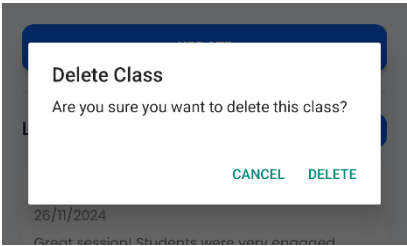


Figure 24. Delete Class Dialog

If deleting class successfully, a toast will be shown and the class is removed from the screen.

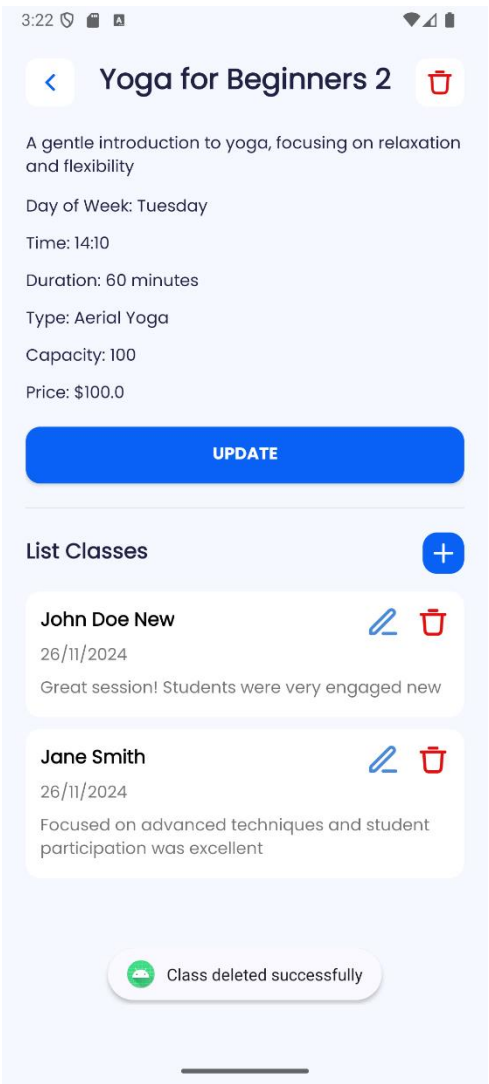


Figure 25. Delete Class Successfully

The data is synchronized from the local to the cloud every period using WorkManager, but if I want to sync it immediately, I can click the sync button on the Course screen to perform the task.

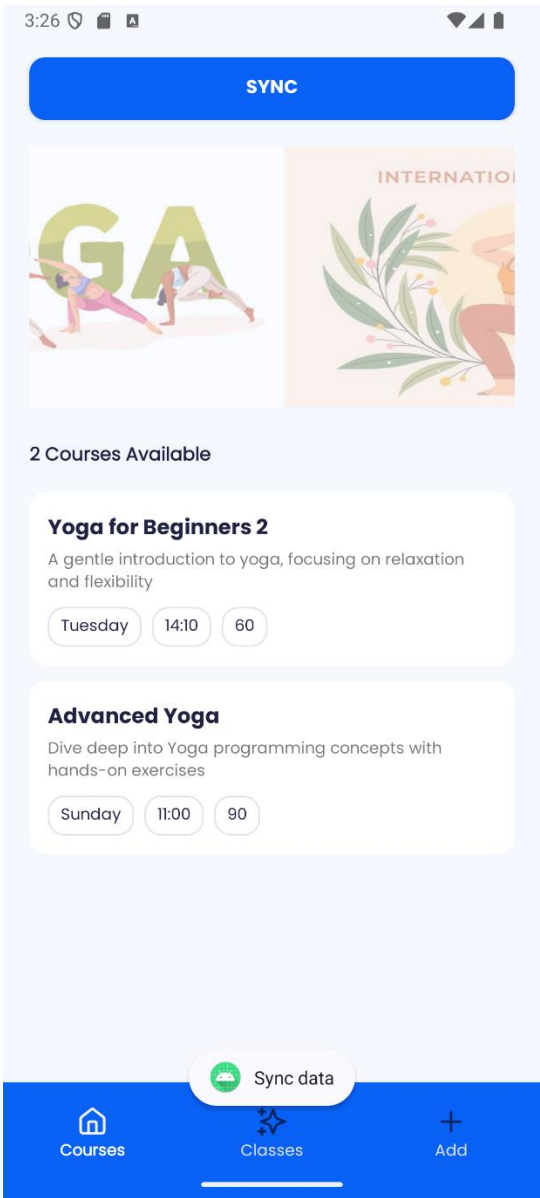


Figure 26. Sync Data to Cloud

The data will be synchronized to the cloud.

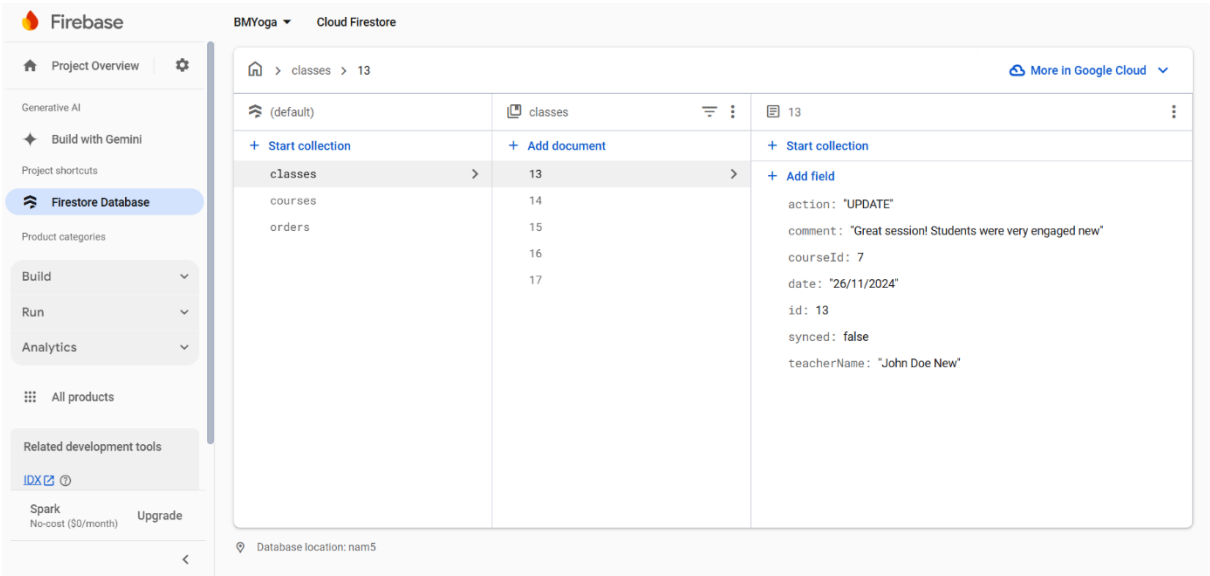


Figure 27. Synchronized Data

2. React Native

In the React Native app, there are three tabs, including Home, which shows a list of classes; Cart, which shows a list of ordered classes, and Profile, which shows the list of booked classes.

On the **Home screen** is a search input with a day of the week selected and then a list of classes.

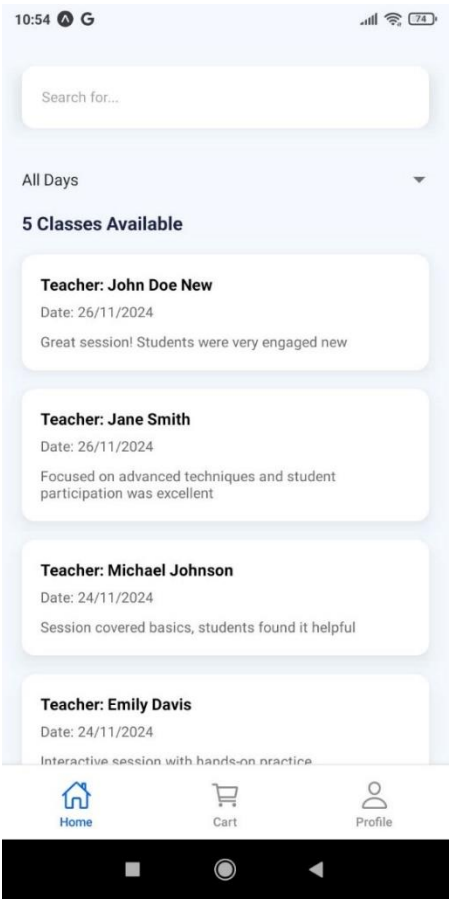


Figure 28. Home Screen

When users type something in searching input and select a day of the week in the select, the results will show classes that match with the filter.

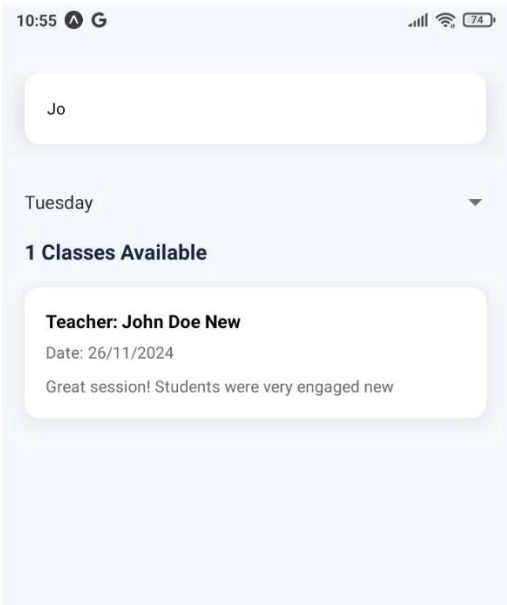


Figure 29. Filtered Classes

When clicking on any class on the **Home screen**, it will navigate to the detail screen of the class, which also includes the course detail with a button to add to the cart.

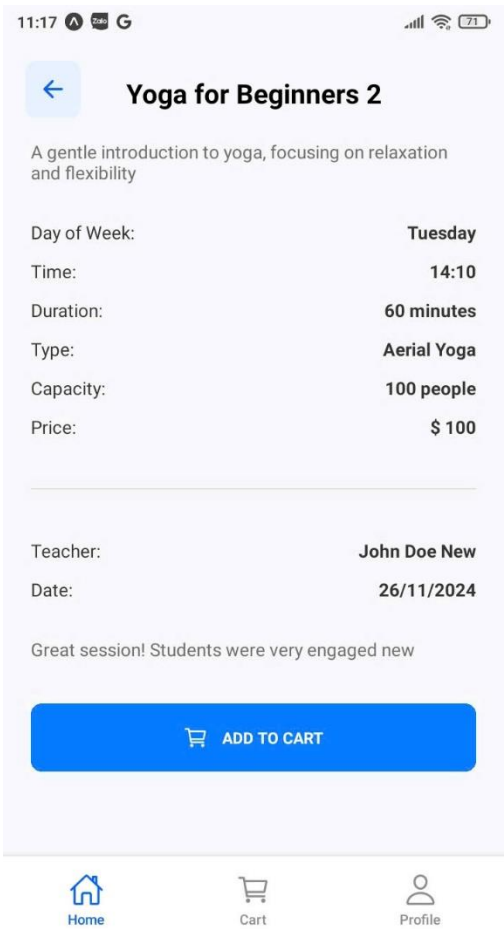


Figure 30. Detail Class Screen

When clicking on the button add to cart, if successful, an alert will be shown to let the user know the current state, but if the user adds the same class to the cart, an alert will show another message.

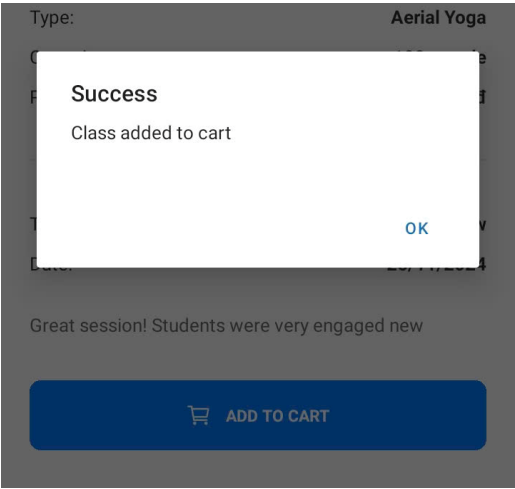


Figure 31. Add to Cart Successfully

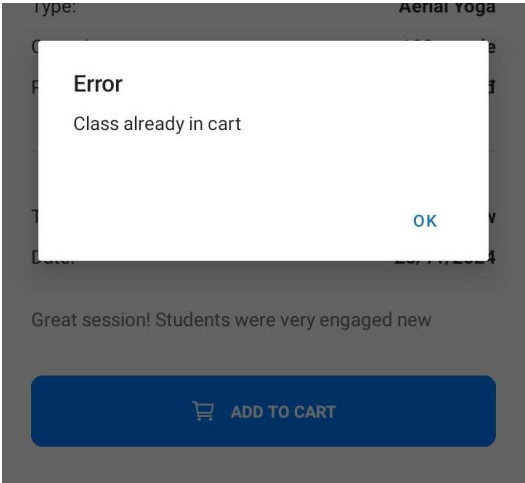


Figure 32. Add to Cart Fail

The **Cart screen** will display a list of ordered classes with input to let the user enter the email for placing an order later.

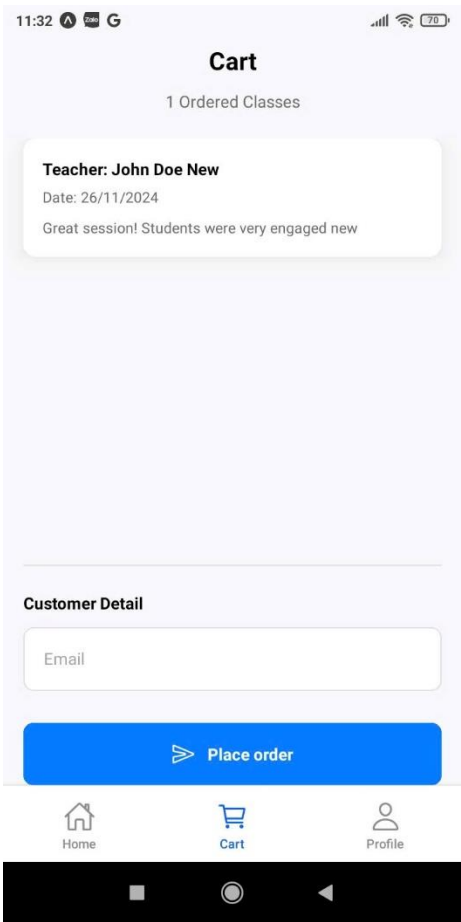


Figure 33. Cart Screen

When a user clicks the button without entering an email address, an alert will be displayed to remind the user to fill in the field.

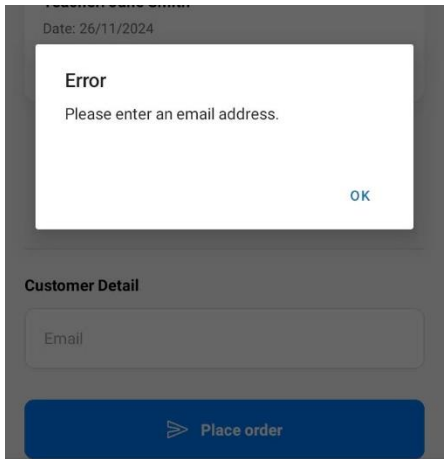


Figure 34. Error Message when Empty Email

If the user enters the email address correctly and clicks the button, the cart will be reset with a success alert, and then the classes will be placed.

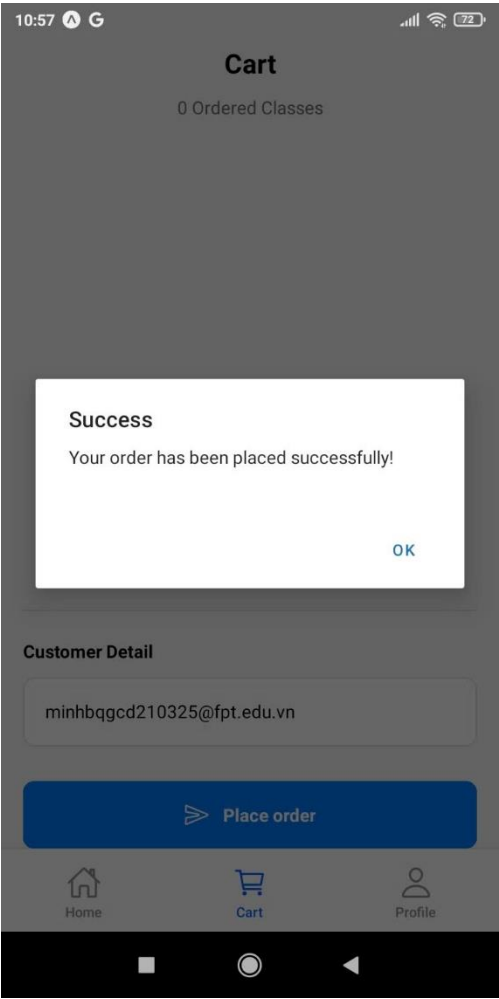


Figure 35. Place Order Successfully

After adding classes to the cart, the class data will be stored in the cloud.

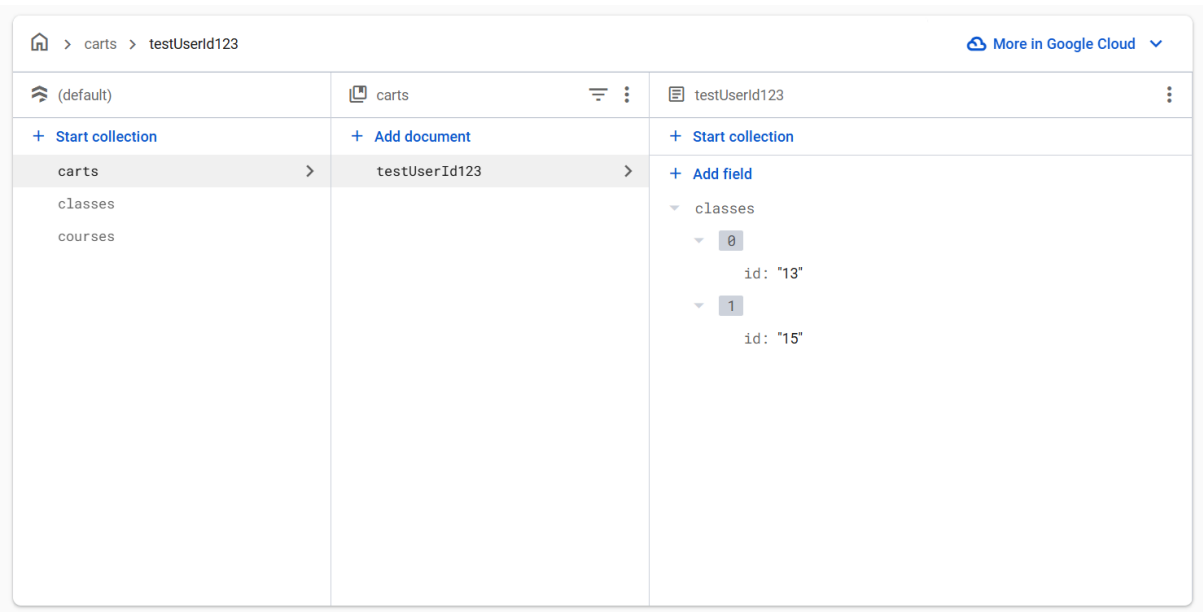


Figure 36. Carts Collection in the Cloud

After placing an order successfully, the user can move to the **Profile screen** to view booked classes.

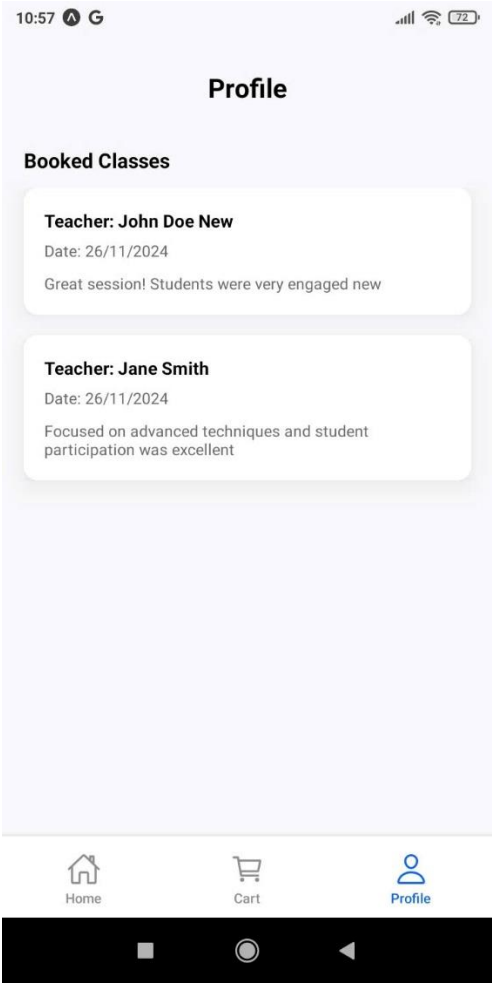


Figure 37. Profile Screen

After placing an order successfully, the class data will be stored in the order collection.

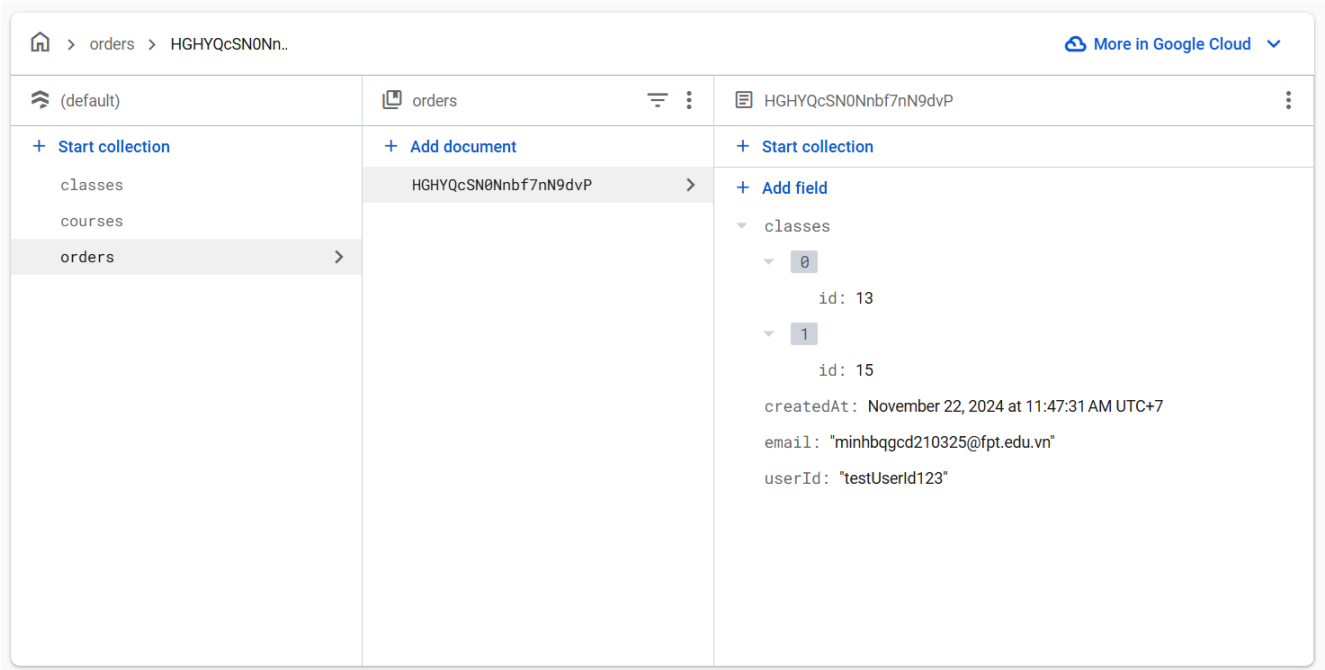


Figure 38. Orders Collection in the Cloud

5. CODE (2%)

1. Android

Architecture

Following the MVVM (Model-View-ViewModel) architectural pattern guarantees a scalable, modular, and clean structure. Working with the ViewModel, which uses LiveData for reactive updates to handle UI-related data and business logic, the Activity/Fragment forms the UI layer. Data is kept locally in an SQLite database under management via Room Database with DAO and Entities. A WorkManager guarantees consistent operations by handling background tasks including data syncing from the local database to the Firebase Cloud.

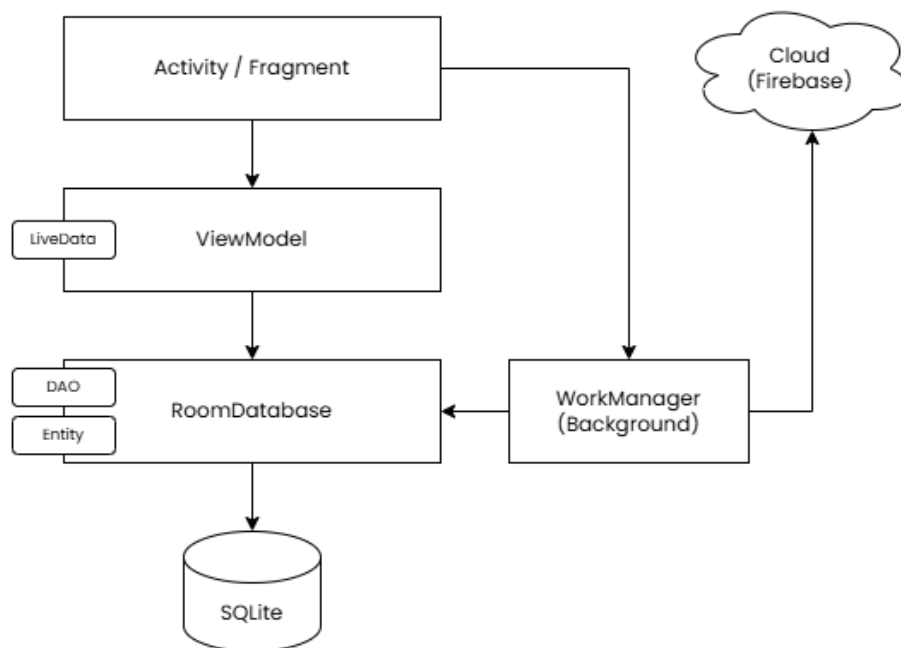


Figure 39. Project's Architecture

Folder Structure (Android)

Firstly, I will explain the code in the Android app; I set up a structural codebase based on different purposes and concerns.

The activities folder contains activities that match some layouts.

The adapters folder contains adapters for displaying a list of items and holders.

The database folder contains SQLite configuration and DAO interfaces belonging to the DAO pattern.

The entities folder contains folders, including enums, which include enum files, and models, which include main model objects for the whole application.

The fragments folder contains three main fragments, including courses, classes, and additions that align with the bottom navigation to make the UI more intuitive.

The utils folder contains reusable methods.

The viewModels folder contains view models that are a part of MVVM architecture.

The worker's folder contains work managers who perform persistent work.

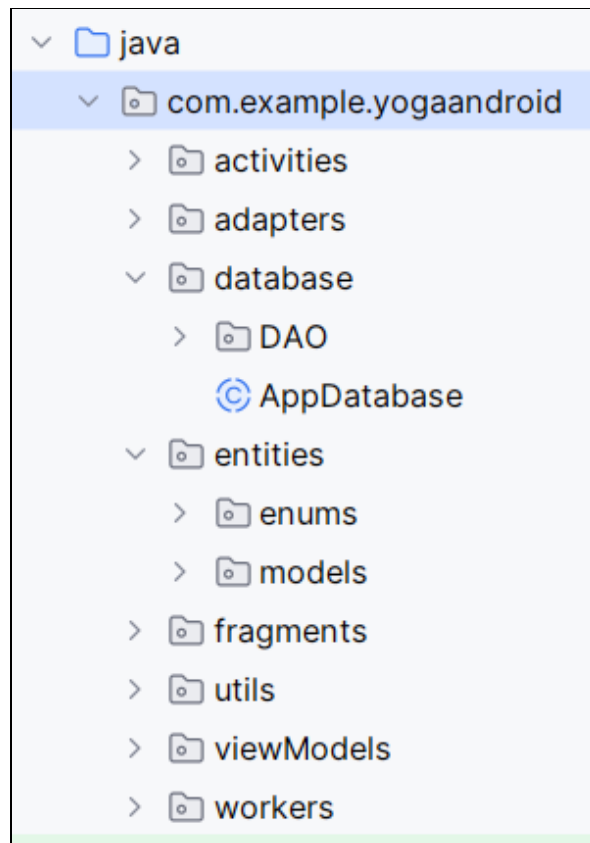


Figure 40. Folder Structure for Android app

Code for adding yoga course (Android):

```
// Set up click listener for the save button
btnSave.setOnClickListener(v -> {
    if (validateInput()) {
        Course course = collectCourseData();
        try {
            courseVM.insertCourse(course);
            Toast.makeText(requireContext(), "Course created successfully",
                Toast.LENGTH_SHORT).show();
            resetFields();
        } catch (Exception e) {
            Toast.makeText(requireContext(), "Error saving course",
                Toast.LENGTH_SHORT).show();
        }
    } else {
        Toast.makeText(requireContext(), "Please fill in all fields",
            Toast.LENGTH_SHORT).show();
    }
});
```

When the user clicks the "Save" button to add a yoga course, the app first validates the input, collects the course details, and tries to save it. Success or error messages are shown after that.


```
// Helper method to validate input
private boolean validateInput() {
    if (etTitle.getText().toString().trim().isEmpty()) {
        etTitle.setError("Title is required");
        return false;
    }
    if (etDescription.getText().toString().trim().isEmpty()) {
        etDescription.setError("Description is required");
        return false;
    }
    if (etPrice.getText().toString().trim().isEmpty()) {
        etPrice.setError("Price is required");
        return false;
    }
    if (etTime.getText().toString().trim().isEmpty()) {
        etTime.setError("Time is required");
        return false;
    }
    if (etDuration.getText().toString().trim().isEmpty()) {
        etDuration.setError("Duration is required");
        return false;
    }
    if (etCapacity.getText().toString().trim().isEmpty()) {
        etCapacity.setError("Capacity is required");
        return false;
    }

    return true;
}
```

This method checks if all required fields (like title, description, price, etc.) are filled. If something is missing, it shows an error message to guide the user.

```
// Helper method to collect course data from EditText fields
private Course collectCourseData() {
    String title = etTitle.getText().toString().trim();
    String description = etDescription.getText().toString().trim();
    double price = Double.parseDouble(etPrice.getText().toString().trim());
    int duration = Integer.parseInt(etDuration.getText().toString().trim());
    int capacity = Integer.parseInt(etCapacity.getText().toString().trim());
    String dayOfWeek = spinnerDayOfWeek.getSelectedItem().toString();
    String time = etTime.getText().toString().trim();
    CourseType type = CourseType.fromId(rgCourseType.getCheckedRadioButtonId());

    return new Course(0, title, description, dayOfWeek, time, duration, capacity,
        price, type, CourseAction.ADD, false);
}
```

This method gathers all the information entered by the user (e.g., title, price, duration) and creates a Course object, ready to be saved.

```
// Insert course into database using DAO
public void insertCourse(Course course) {
    courseDao.insertCourse(course);
}
```

This method takes the Course object and passes it to the DAO to save it in the database.

```
@Dao
public interface CourseDao {
    @Insert
    void insertCourse(Course course);
}
```

Defines the insertCourse method, which uses Room's @Insert annotation to handle saving the course to the database automatically.

Code for viewing list of yoga class details (Android):

```
// Initialize the ViewModel
coursesVM = new ViewModelProvider(requireActivity()).get(CoursesVM.class);

// Observe LiveData
coursesVM.getCourses().observe(getViewLifecycleOwner(), courses -> {
    courseAdapter.setCourses(courses);
    numCourses.setText(courses.size() + " Courses Available");
});
```

Handles ViewModel initialization and observes course data to update the UI dynamically.

```
public class CoursesVM extends AndroidViewModel {
    private final MutableLiveData<List<Course>> courses;
    private final CourseDao courseDao;

    public CoursesVM(Application application) {
        super(application);
        courseDao = AppDatabase.getDatabase(application).courseDao();
        courses = new MutableLiveData<>();
        loadCourses(); // Preload the courses data on ViewModel creation
    }

    // Load all courses into LiveData
    private void loadCourses() {
        List<Course> courseList = courseDao.getAllCourses(CourseAction.DELETE); //
        Fetch from the database
        courses.setValue(courseList);
    }

    // Methods for returning LiveData
    public LiveData<List<Course>> getCourses() {
        return courses;
    }
}
```

Manages course data in the ViewModel, preloading it from the database and providing it as LiveData.

```
@Query("SELECT * FROM courses WHERE `action` != :deleteAction")
List<Course> getAllCourses(CourseAction deleteAction);
```

Defines a query to fetch all courses except those marked for deletion.

Code for viewing a yoga class details (Android):

```
private void loadCourseDetails(int courseId) {
    Course course = courseDetailVM.getCourseById(courseId);
    if (course != null) {
        tvCourseName.setText(course.getName());
        tvCourseDescription.setText(course.getDescription());
        tvDayOfWeek.setText("Day of Week: " + course.getDayOfWeek());
        tvTime.setText("Time: " + course.getTime());
        tvDuration.setText("Duration: " + course.getDuration() + " minutes");
        tvType.setText("Type: " + course.getType().getName());
        tvCapacity.setText("Capacity: " + course.getCapacity());
        tvPrice.setText("Price: $" + course.getPrice());

        dateOfWeek = course.getDayOfWeek();
    } else {
        Toast.makeText(this, "Course not found", Toast.LENGTH_SHORT).show();
    }
}
```

Loads and displays the details of a specific course by its ID, or shows a message if the course is not

```
public Course getCourseById(int courseId) {
    return courseDao.getCourseById(courseId, CourseAction.DELETE);
}
```

Fetches a course by its ID from the database using the DAO while excluding deleted courses.

```
@Query("SELECT * FROM courses WHERE id = :courseId AND `action` != :deleteAction")
Course getCourseById(int courseId, CourseAction deleteAction);
```

Defines a query to retrieve a course by its ID, filtering out courses marked for deletion.

Code for updating a yoga class (Android):

```
private void loadCourseData() {
    Course course = courseVM.getCourseById(courseId);
    if (course == null) {
        Toast.makeText(this, "Course not found", Toast.LENGTH_SHORT).show();
        return;
    }
    etTitle.setText(course.getName());
    etDescription.setText(course.getDescription());
    etPrice.setText(String.valueOf(course.getPrice()));
    etDuration.setText(String.valueOf(course.getDuration()));
    etCapacity.setText(String.valueOf(course.getCapacity()));
    etTime.setText(course.getTime());
    rgCourseType.check(course.getType().getId());
    spinnerDayOfWeek.setSelection(getSpinnerIndex(course.getDayOfWeek()));
}

// Helper method to get the index of a value in the Spinner
private int getSpinnerIndex(String value) {
    for (int i = 0; i < spinnerDayOfWeek.getCount(); i++) {
```

```

        if
(spinnerDayOfWeek.getItemAtPosition(i).toString().equalsIgnoreCase(value)) {
            return i;
        }
    }
    return 0;
}

private void updateCourse() {
    if (courseId == -1) {
        Toast.makeText(this, "Invalid course ID", Toast.LENGTH_SHORT).show();
        return;
    }

    if (validateInput()) {
        Course course = collectCourseData();
        try {
            courseVM.updateCourse(course);
            Toast.makeText(this, "Course updated successfully",
Toast.LENGTH_SHORT).show();
            setResult(RESULT_OK);
            finish();
        } catch (Exception e) {
            Toast.makeText(this, "Error saving course",
Toast.LENGTH_SHORT).show();
        }
    }
}
}

```

Loads existing course data into the input fields for editing and updates the course in the database if the input is valid.

```

public void updateCourse(Course course) {
    courseDao.updateCourse(course);
}

```

Sends the updated course data to the DAO for saving changes in the database.

```

@Update
void updateCourse(Course course);

```

Defines an @Update query to update an existing course record in the database.

Code for deleting a yoga class (Android):

```

private void showDeleteConfirmationDialog() {
    new AlertDialog.Builder(this)
        .setTitle("Delete Course")
        .setMessage("Are you sure you want to delete this course?")
        .setPositiveButton("Delete", (dialog, which) -> {
            courseDetailVM.deleteCourse(courseId);
            Toast.makeText(this, "Course deleted successfully",
Toast.LENGTH_SHORT).show();
            finish();
        })
        .setNegativeButton("Cancel", (dialog, which) -> dialog.dismiss())
}

```

```

        .show();
    }
}

```

Displays a confirmation dialog to the user before deleting the course, and deletes the course if confirmed.

```

public void deleteCourse(int courseId) {
    courseDao.deleteCourse(courseId, CourseAction.DELETE);
    List<ClassSession> classesToDelete = classDao.getClassesForCourse(courseId,
        ClassAction.DELETE);
    for (ClassSession classSession : classesToDelete) {
        classDao.deleteClass(classSession.getId(), ClassAction.DELETE);
    }
}

```

Deletes the course and its associated class sessions from the database by marking them as deleted.

Code for managing class instance (Android):

```

// Helper method to show the upsert class dialog
private void showUpsertClassDialog(ClassSession classSession) {
    final Dialog dialog = new Dialog(this);
    dialog.requestWindowFeature(Window.FEATURE_NO_TITLE);
    dialog setContentView(R.layout.dialog_upsert_class);

    Window window = dialog.getWindow();
    if (window == null) return;

    window.setLayout(WindowManager.LayoutParams.MATCH_PARENT,
        WindowManager.LayoutParams.WRAP_CONTENT);
    window.setBackgroundDrawable(new ColorDrawable(Color.TRANSPARENT));

    WindowManager.LayoutParams windowAttributes = window.getAttributes();
    windowAttributes.gravity = Gravity.BOTTOM;
    window.setAttributes(windowAttributes);

    dialog.setCancelable(true);

    // Initialize dialog views
    TextView tvTitle = dialog.findViewById(R.id.tvTitleClass);
    EditText etTitle = dialog.findViewById(R.id.etTitleClass);
    EditText etDate = dialog.findViewById(R.id.etDateClass);
    EditText etComment = dialog.findViewById(R.id.etCommentClass);
    Button btnCancel = dialog.findViewById(R.id.btnCancelClass);
    Button btnSave = dialog.findViewById(R.id.btnSaveClass);

    // If classSession is not null, update the dialog
    if (classSession != null) {
        tvTitle.setText("UPDATE CLASS");
        btnSave.setText("UPDATE");
        etTitle.setText(classSession.getTeacherName());
        etDate.setText(classSession.getDate());
        etComment.setText(classSession.getComment());
    }

    // Handle date picker
}

```

```

etDate.setOnClickListener(view -> {
    final Calendar calendar = Calendar.getInstance();
    int year = calendar.get(Calendar.YEAR);
    int month = calendar.get(Calendar.MONTH);
    int day = calendar.get(Calendar.DAY_OF_MONTH);

    @SuppressWarnings("SetTextI18n") DatePickerDialog datePickerDialog = new
DatePickerDialog(
    this,
    R.style.DatePickerDialogTheme,
    (view1, selectedYear, selectedMonth, selectedDay) -> {
        Calendar selectedDate = Calendar.getInstance();
        selectedDate.set(selectedYear, selectedMonth, selectedDay);

        // Check if the selected date matches the required day of the
week
        String selectedDayOfWeek = new SimpleDateFormat("EEEE",
Locale.getDefault()).format(selectedDate.getTime());

        if (!selectedDayOfWeek.equalsIgnoreCase(dateOfWeek)) {
            etDate.setError("Selected date must be a " + dateOfWeek);
            Toast.makeText(this, "Please select a " + dateOfWeek + "
class", Toast.LENGTH_SHORT).show();
        } else {
            etDate.setText(selectedDay + "/" + (selectedMonth + 1) +
"/" + selectedYear);
            etDate.setError(null); // Clear any previous errors
        }
    }, year, month, day);

    datePickerDialog.getDatePicker().setMinDate(calendar.getTimeInMillis());
    datePickerDialog.show();
});

// Handle cancel button click
btnCancel.setOnClickListener(view -> dialog.dismiss());

// Handle save button click
btnSave.setOnClickListener(view -> {
    // Get input values
    String teacherName = etTitle.getText().toString().trim();
    String date = etDate.getText().toString().trim();
    String comment = etComment.getText().toString().trim();

    if (teacherName.isEmpty()) {
        etTitle.setError("Teacher name is required");
        return;
    }

    if (date.isEmpty()) {
        etDate.setError("Date is required");
        return;
    }

    if (classSession != null) {
        // Update existing class session
        classSession.setTeacherName(teacherName);
        classSession.setDate(date);
        classSession.setComment(comment);
        classSession.setAction(ClassAction.UPDATE);
        classSession.setSynced(false);
        courseDetailVM.updateClassSession(classSession);
        courseDetailVM.loadClasses(courseId);
        Toast.makeText(this, "Class updated successfully",

```

```

Toast.LENGTH_SHORT).show();
    } else {
        // Add a new class session
        ClassSession newClassSession = new ClassSession(teacherName, date,
comment, courseId, ClassAction.ADD, false);
        courseDetailVM.insertClassSession(newClassSession);
        courseDetailVM.loadClasses(courseId);
        Toast.makeText(this, "Class added successfully",
Toast.LENGTH_SHORT).show();
    }

    dialog.dismiss();
});

dialog.show();
}

```

This function displays a dialog for adding or updating a class session, handles date selection, validates input, and communicates the action to the ViewModel for persistence.

```

private void showClassDeleteConfirmation(int classId) {
    new AlertDialog.Builder(this)
        .setTitle("Delete Class")
        .setMessage("Are you sure you want to delete this class?")
        .setPositiveButton("Delete", (dialog, which) -> {
            courseDetailVM.deleteClassSession(classId); // Delete the class
session
            courseDetailVM.loadClasses(courseId);
            Toast.makeText(this, "Class deleted successfully",
Toast.LENGTH_SHORT).show();
        })
        .setNegativeButton("Cancel", (dialog, which) -> dialog.dismiss())
        .show();
}

```

Shows a confirmation dialog before deleting a class session and handles the deletion process if confirmed.

```

public void loadClasses(int courseId) {
    classes.setValue(classDao.getClassesForCourse(courseId, ClassAction.DELETE));
}

public void insertClassSession(ClassSession classSession) {
    classDao.insertClass(classSession);
}

public void updateClassSession(ClassSession classSession) {
    classDao.updateClass(classSession);
}

public void deleteClassSession(int classSessionId) {
    classDao.deleteClass(classSessionId, ClassAction.DELETE);
}

```

Provides methods for inserting, updating, and deleting class sessions in the database, and loads the current list of class sessions for a course into LiveData.

Code for searching class instances (Android):

```
searchInput.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence charSequence, int i, int i1, int
i2) { }
    @Override
    public void onTextChanged(CharSequence charSequence, int i, int i1, int i2)
{ }
    @Override
    public void afterTextChanged(Editable editable) {
        // After the text has been changed, filter the classes by teacher's name.
        String searchTerm = editable.toString().trim();
        classesVM.getClassesFiltered(selectedDay, searchTerm);
    }
});
```

Filters class sessions by the teacher's name as the user types in the search input.

```
private void selectDay(TextView selectedView) {
    if (selectedDayView != null) {
        // Reset previously selected day style
        selectedDayView.setBackgroundResource(R.drawable.day_button_background);
        selectedDayView.setTextColor(Color.parseColor("#333333"));
    }

    // Set new selected day style
    selectedDayView = selectedView;

    selectedDayView.setBackgroundResource(R.drawable.selected_day_button_background);
    selectedDayView.setTextColor(Color.WHITE);

    // Get the selected day as a string
    selectedDay = selectedView.getText().toString();

    String searchTerm = searchInput.getText().toString();
    classesVM.getClassesFiltered(selectedDay, searchTerm);
}
```

Updates the selected day visually, applies styles to indicate the selection, and filters classes based on the selected day and current search term.

```
// Filter Classes by teacher name
public void filterClasses(String searchTerm) {
    List<ClassSession> filteredClasses = classDao.getClassesFiltered(searchTerm,
ClassAction.DELETE);
    classes.setValue(filteredClasses);
}

// Filter by date of week based on current classes not from DAO
public void getClassesFiltered(String selectedDay, String searchTerm) {
```



```

filterClasses(searchTerm);

// Retrieve the current list of classes from LiveData
List<ClassSession> currentClasses = classes.getValue();
if (classes == null) return;

List<ClassSession> filteredClasses = new ArrayList<>();
for (ClassSession classSession : currentClasses) {
    try {
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(dateFormat.parse(classSession.getDate()));

        // Get the day of the week as a string (e.g., "Monday")
        String dayOfWeek = calendar.getDisplayName(Calendar.DAY_OF_WEEK,
Calendar.LONG, Locale.getDefault());

        // Check if the day matches the selected day or if "All" is selected
        if (dayOfWeek.equalsIgnoreCase(selectedDay) ||
selectedDay.equalsIgnoreCase("All")) {
            filteredClasses.add(classSession);
        }
    } catch (ParseException e) {
        e.printStackTrace();
    }
}
classes.setValue(filteredClasses); // Update LiveData with the filtered list
}

```

filterClasses: Filters class sessions by teacher's name using the database.

getClassesFiltered: Filters class sessions based on the selected day of the week and updates the LiveData with the results.

Code for uploading details to a cloud-based web service (Android):

```

public class SyncWorker extends Worker {
    private final CourseDao courseDao;
    private final ClassDao classSessionDao;
    private final FirebaseFirestore firestore;

    public SyncWorker(@NonNull Context context, @NonNull WorkerParameters
workerParams) {
        super(context, workerParams);
        AppDatabase db = AppDatabase.getDatabase(context);
        courseDao = db.courseDao();
        classSessionDao = db.classDao();
        firestore = FirebaseFirestore.getInstance(); // Initialize Firestore
    }

    @NonNull
    @Override
    public Result doWork() {
        // Sync Courses
        syncCourses();

        // Sync ClassSessions
        syncClassSessions();

        return Result.success();
    }
}

```

```

// Sync Courses
private void syncCourses() {
    List<Course> unsyncedCourses = courseDao.getUnsyncedCourses();
    for (Course course : unsyncedCourses) {
        switch (course.getAction()) {
            case ADD:
                syncUpsertCourse(course);
                break;
            case UPDATE:
                syncUpsertCourse(course);
                break;
            case DELETE:
                syncDeleteCourse(course);
                break;
        }
    }
}

private void syncUpsertCourse(Course course) {
    firestore.collection("courses").document(String.valueOf(course.getId()))
        .set(course)
        .addOnSuccessListener(aVoid -> {
            // Mark as synced after successful upsert
            courseDao.updateCourseSyncStatus(course.getId(), true);
        })
        .addOnFailureListener(e -> {
            Log.e("SyncWorker", "Error syncing new course: " +
course.getId(), e);
        });
}

private void syncDeleteCourse(Course course) {
    firestore.collection("courses").document(String.valueOf(course.getId()))
        .delete()
        .addOnSuccessListener(aVoid -> {
            // Delete course locally after successful deletion in
Firestore
            courseDao.deleteCourseById(course.getId());
        })
        .addOnFailureListener(e -> {
            Log.e("SyncWorker", "Error deleting course: " +
course.getId(), e);
        });
}

private void syncClassSessions() {
    List<ClassSession> unsyncedClasses = classSessionDao.getUnsyncedClasses();
    for (ClassSession classSession : unsyncedClasses) {
        switch (classSession.getAction()) {
            case ADD:
                syncUpsertClass(classSession);
                break;
            case UPDATE:
                syncUpsertClass(classSession);
                break;
            case DELETE:
                syncDeleteClass(classSession);
                break;
        }
    }
}

private void syncUpsertClass(ClassSession classSession) {

```

```

    firestore.collection("classes").document(String.valueOf(classSession.getId()))
        .set(classSession)
        .addOnSuccessListener(aVoid -> {
            // Mark as synced after successful upsert
            classSessionDao.updateClassSyncStatus(classSession.getId());
        })
        .addOnFailureListener(e -> {
            Log.e("SyncWorker", "Error syncing new class session: " +
classSession.getId(), e);
        });
    }

    private void syncDeleteClass(ClassSession classSession) {
        firestore.collection("classes").document(String.valueOf(classSession.getId()))
            .delete()
            .addOnSuccessListener(aVoid -> {
                // Delete class locally after successful deletion in Firestore
                classSessionDao.deleteClassById(classSession.getId());
            })
            .addOnFailureListener(e -> {
                Log.e("SyncWorker", "Error deleting class session: " +
classSession.getId(), e);
            });
    }
}

```

Implements a background worker to sync local course and class session data with Firestore by handling ADD, UPDATE, and DELETE operations.

In the class, my lecture recommended many ways to sync the data to the cloud, and I really appreciate that, but I think in production, these sync tasks need to be set periodically like every 12 hours should be better, and this is long-running.

```

@Query("SELECT * FROM courses WHERE isSynced = 0") // Fetch unsynced data
List<Course> getUnsyncedCourses();

@Query("SELECT * FROM classes WHERE isSynced = 0") // Fetch unsynced data
List<ClassSession> getUnsyncedClasses();

@Query("UPDATE courses SET isSynced = :isSynced WHERE id = :id")
void updateCourseSyncStatus(int id, boolean isSynced);

@Query("DELETE FROM courses WHERE id = :id")
void deleteCourseById(int id);

@Query("UPDATE classes SET isSynced = 1 WHERE id = :classId")
void updateClassSyncStatus(int classId);

@Query("DELETE FROM classes WHERE id = :classId")
void deleteClassById(int classId);

```

Defines SQL queries for fetching unsynced records, updating sync status, and deleting local records in the database.

```

schedulePeriodicSyncWork(requireContext());

btnSync.setOnClickListener(v ->
{
    Toast.makeText(requireContext(), "Sync data", Toast.LENGTH_SHORT).show();
    triggerOneTimeSyncWork(requireContext());
});

private void schedulePeriodicSyncWork(Context context) {
    Constraints constraints = new Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED)
        .build();

    PeriodicWorkRequest periodicSyncWorkRequest = new
PeriodicWorkRequest.Builder(SyncWorker.class, 15, TimeUnit.MINUTES)
        .setConstraints(constraints)
        .build();

    WorkManager.getInstance(context).enqueueUniquePeriodicWork(
        "PeriodicSyncWork",
        ExistingPeriodicWorkPolicy.KEEP,
        periodicSyncWorkRequest
    );
}

// Helper method to trigger one-time sync
private void triggerOneTimeSyncWork(Context context) {
    Constraints constraints = new Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED)
        .build();

    OneTimeWorkRequest oneTimeSyncWorkRequest = new
OneTimeWorkRequest.Builder(SyncWorker.class)
        .setConstraints(constraints)
        .build();

    WorkManager.getInstance(context).enqueue(oneTimeSyncWorkRequest);
}

```

This code schedules periodic background syncing every 15 minutes using WorkManager and provides a button to trigger a one-time sync immediately, ensuring data is synchronized with Firestore only when the device is connected to a network.

Code for Carousel (Android):

```

if (isConnected(requireContext())) {
    ActionViewFlipper();
} else {
    Toast.makeText(requireContext(), "No Internet", Toast.LENGTH_LONG).show();
}

private void ActionViewFlipper() {
    // Create a list of image URLs for the advertisements
    List<String> mangquangcao = new ArrayList<>();

    mangquangcao.add("https://t4.ftcdn.net/jpg/07/11/73/49/360_F_711734922_3XfZ5aDzqE8Pb8G1FAFYylb46BqXKQXQ.jpg");
    mangquangcao.add("https://bizweb.dktcdn.net/100/514/950/products/cover-khi-nao-ban-khong-can-tap-yoga.jpg?v=1717648220867");
    mangquangcao.add("https://img.freepik.com/free-vector/flat-background-international-yoga-day-celebration_23-2150388305.jpg");
}

```

```

    for (int i = 0; i < mangquangcao.size(); i++) {
        ImageView imageView = new ImageView(requireContext());
        Glide.with(requireContext()).load(mangquangcao.get(i)).into(imageView);
        Log.d("CoursesFragment", "Adding image to flipper: " +
mangquangcao.get(i));
        imageView.setScaleType(ImageView.ScaleType.FIT_XY);
        viewFlipper.addView(imageView);
    }
    viewFlipper.setFlipInterval(3000);
    viewFlipper.setAutoStart(true);
    Animation slide_in = AnimationUtils.loadAnimation(requireContext(),
R.anim.slide_in_right);
    Animation slide_out = AnimationUtils.loadAnimation(requireContext(),
R.anim.slide_out_right);
    viewFlipper.setInAnimation(slide_in);
    viewFlipper.setOutAnimation(slide_out);
}

```

This code checks for an active internet connection; if available, it initializes a ViewFlipper to display a slideshow of advertisement images with animations using URLs loaded via Glide. If no connection is detected, it shows a toast message indicating "No Internet." In the future, it would be better if there is a place to store asset images like this so the admin can update images in real-time and not fixed like this.

2. React Native

Folder Structure (React Native)

The assets folder is used to store stylesheets, fake data to test, fonts, and images.

The components folder contains reusable components that are used in the whole application.

The config folder contains configurations, but now I just store the Firebase configuration.

The context folder contains global data that can be used in the whole application without passing props down manually.

The models folder contains types for main objects, which is very familiar when using TypeScript.

The navigators folder contains the configuration for stack navigation.

The pages folder contains the main screens for the application.

The services folder contains services for each model to interact with Firebase.

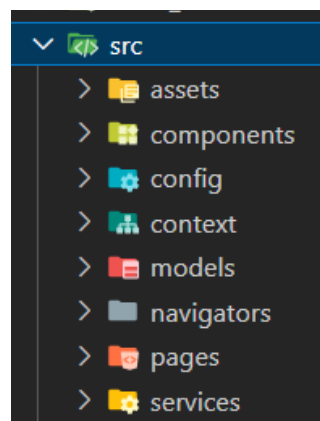


Figure 41. Folder Structure for React Native

Code for displaying the list of classes (React Native)

```
const [allClasses, setAllClasses] = useState<ClassSession[]>([]);

<FlatList
  data={classes}
  keyExtractor={(item) => item.id}
  renderItem={({ item }) => <ClassItem item={item} />}
  contentContainerStyle={styles.classList}
/>

export const fetchClasses = async (): Promise<ClassSession[]> => {
  try {
    const classesCollection = collection(db, "classes");
    const classSnapshot = await getDocs(classesCollection);
    const classList = classSnapshot.docs.map((doc) => {
      const data = doc.data();
      return {
        id: doc.id,
        teacherName: data.teacherName,
        date: data.date,
        comment: data.comment,
        courseId: data.courseId,
      } as ClassSession;
    });

    return classList;
  } catch (error) {
    return [];
  }
}
```

This code defines a `fetchClasses` function to fetch class data from Firestore, returning a list of `ClassSession` objects. It uses `FlatList` in a React component to display the fetched classes, where each item is rendered using the `ClassItem` component.

Code for adding class to cart (React Native)

```
const handleAddToCart = async () => {
  if (!userId || !classDetail) {
    Alert.alert("Error", "User ID or class details not available.");
    return;
  }

  const response = await addToCart(userId, {
    id: classDetail.id,
  });
}
```

```

    Alert.alert(response.success ? "Success" : "Error", response.message);
  };

export async function addToCart(userId: string, classData: { id: string }) {
  try {
    const cartRef = doc(db, "carts", userId);
    const cartSnap = await getDoc(cartRef);

    if (cartSnap.exists()) {
      // Cart exists, check if the class is already in the cart
      const cartData = cartSnap.data();
      const classExists = cartData.classes.some(
        (cartClass: { id: string }) => cartClass.id === classData.id
      );

      if (classExists) {
        return { success: false, message: "Class already in cart" };
      }

      // Add only the class id to the cart using arrayUnion (to avoid duplicates)
      await updateDoc(cartRef, {
        classes: arrayUnion({ id: classData.id }), // Store only the class id
      });
    } else {
      // If the cart document doesn't exist, create it with the class id
      await setDoc(cartRef, { classes: [{ id: classData.id }] }); // Store only the
class id
    }

    return { success: true, message: "Class added to cart" };
  } catch (error) {
    return { success: false, message: "Error adding to cart" };
  }
}

```

This code defines a `handleAddToCart` function that checks if the `userId` and `classDetail` are available, and if so, it calls the `addToCart` function to add a class to the user's cart in Firestore. The `addToCart` function either updates an existing cart with the new class or creates a new cart document if one doesn't exist, ensuring no duplicate classes are added. It returns a success or error message based on the outcome.

Code for placing order (React Native)

```

const handlePlaceOrder = async () => {
  if (!email) {
    Alert.alert("Error", "Please enter an email address.");
    return;
  }
}

```

```

if (orderedClasses.length === 0) {
  Alert.alert("Error", "Your cart is empty.");
  return;
}

if (!userId) {
  Alert.alert("Error", "User ID is missing.");
  return;
}

const orderResult = await placeOrder(
  userId,
  email,
  orderedClasses.map((cls) => ({ id: cls.id })))
);

if (orderResult.success) {
  Alert.alert("Success", "Your order has been placed successfully!");
  setOrderedClasses([]);
} else {
  Alert.alert("Error", "Failed to place order.");
}
};

export const placeOrder = async (userId: string, email: string, orderedClasses: { id:
string }[]) => {
  try {
    const orderRef = doc(collection(db, "orders"));
    const orderData = {
      userId: userId,
      email: email,
      classes: orderedClasses,
      createdAt: new Date(),
    };
    // Save the order to Firestore
    await setDoc(orderRef, orderData);

    // Now remove the classes from the cart
    const cartRef = doc(db, "carts", userId);
    await updateDoc(cartRef, {
      classes: arrayRemove(...orderedClasses), // Remove the ordered classes
    });

    await deleteDoc(cartRef);
    return { success: true, message: "Order placed successfully" };
  } catch (error) {
    return { success: false, message: "Error placing order" };
  }
};

```


This code defines a `handlePlaceOrder` function that checks for necessary conditions (email, cart contents, user ID) before calling the `placeOrder` function, which saves the order to Firestore and removes the ordered classes from the user's cart. If the order is successful, it clears the cart and shows a success message; otherwise, it displays an error alert.

Table of Figures

Figure 1. Design Sketching	7
Figure 2. Figma Design	7
Figure 3. res Folder (1)	8
Figure 4. res Folder (2)	8
Figure 5. Courses Screen	9
Figure 6. Course Detail Screen	10
Figure 7. Classes Screen	11
Figure 8. Filtered Classes	12
Figure 9. Add Screen.....	13
Figure 10. Time Picker	13
Figure 11. Errors without Fulfill Information	14
Figure 12. Adding Course Successfully	15
Figure 13. Update Course Screen.....	16
Figure 14. Error Message when Updating.....	17
Figure 15. Update Course Successfully	18
Figure 16. Delete Confirm Dialog for Course	19
Figure 17. Delete Course Successfully.....	19
Figure 18. Add Class Dialog	20
Figure 19. Error Messages when Adding Class.....	20
Figure 20. Date Picker	21
Figure 21. Add Class Successfully	21
Figure 22. Update Class Dialog.....	22
Figure 23. Update Class Successfully.....	22
Figure 24. Delete Class Dialog	23
Figure 25. Delete Class Successfully	23
Figure 26. Sync Data to Cloud	24
Figure 27. Synchronized Data.....	25
Figure 28. Home Screen	25
Figure 29. Filtered Classes	26
Figure 30. Detail Class Screen.....	26
Figure 31. Add to Cart Successfully	27
Figure 32. Add to Cart Fail.....	27
Figure 33. Cart Screen	28
Figure 34. Error Message when Empty Email	28
Figure 35. Place Order Successfully.....	29
Figure 36. Carts Collection in the Cloud	29
Figure 37. Profile Screen	30
Figure 38. Orders Collection in the Cloud	30
Figure 39. Project's Architecture.....	31
Figure 40. Folder Structure for Android app.....	32
Figure 41. Folder Structure for React Native	46

Reference List

Apple, 2023. *Human interface guidelines*. [Online]

Available at: <https://developer.apple.com/design/human-interface-guidelines/>

[Accessed 27 November 2024].

Elmasri, R. and Navathe, S.B., 2015. *Fundamentals of database systems*. 7th ed. Pearson.

Google, 2023. *Material design guidelines*. [Online]

Available at: <https://material.io/>

[Accessed 27 November 2024].

Nielsen, J., 1994. *10 usability heuristics for user interface design*. [Online]

Available at: <https://www.nngroup.com/articles/ten-usability-heuristics/>

[Accessed 27 November 2024].

OWASP, 2023. *Mobile security testing guide*. [Online]

Available at: <https://owasp.org/www-project-mobile-security-testing-guide/>

[Accessed 27 November 2024].

Shneiderman, B., 1998. *Designing the user interface: strategies for effective human-computer interaction*. 3rd ed. Addison-Wesley.