

SINGIDUNUM UNIVERSITY
Software and Information Engineering

**Development of LessonLynx, an Application for Managing Foreign
Language Tuition**

- bachelor's thesis -

*English translation of a thesis originally written and defended in Serbian at Singidunum
University.*

Supervisor:
Prof. Dr Miodrag Živković

Candidate:
Nikola Đermanović

Belgrade, 2026

Contents

Introduction	3
Research Problem	3
Subject of the Research	3
Research Objectives	3
Significance of the Thesis	4
Research Methods	4
Structure of the Thesis	5
1. Overview of Technologies	6
1.1. React Native and Expo	6
1.2. TypeScript	6
1.3. Firebase, Authentication and Firestore	6
1.4. React Navigation	7
1.5. External API, ExchangeRate-API	7
1.6. Other Libraries Used	8
2. System Analysis and Design	9
2.1. Requirements Analysis	9
Students Module	9
Lessons Module	9
Payments Module	9
Dashboard Module (overview/statistics)	9
2.2. Application Architecture	11
2.3. Data Model	13
2.4. Navigation Flow and UI/UX Design	16
3. Implementation	18
3.1. Authentication	18
3.2. Students Module	22
3.3. Lessons Module	26
3.4. Payments Module and FIFO Logic	30
3.5. Dashboard and External API	33
4. Testing	36
4.1. Functional Testing	36
4.2. Authentication Testing and Known Issues	40
Concluding Remarks	42
References	46

Introduction

Research Problem

With the development of digital technologies and the growing popularity of learning foreign languages online, an increasing number of teachers of English as a foreign language (*ESL, English as a Second Language*) are choosing to stop working through online schools and platforms and to work independently. The reason for such a step is most often better working conditions: higher earnings, more flexible working hours and a direct relationship with students, without an intermediary that retains part of the income. However, the transition to independent work places the teacher in a position similar to that of an entrepreneur, where, in addition to teaching itself, they must also handle administrative tasks on their own: keeping records of lessons held, tracking student debts and calculating monthly earnings, often in several different currencies. Keeping such records manually, most commonly by means of spreadsheets (e.g. Excel or Google Sheets), becomes difficult to manage and prone to errors as the number of students grows, and the process of calculating payments and tracking unpaid lessons is made even harder when dealing with students from different countries who pay in different currencies.

The topicality of this subject is further confirmed by the spread of the BaaS (Backend as a Service) approach in software development, which allows independent developers and small teams to use functionalities such as authentication and data storage as a ready-made service, without the need to develop and maintain their own server infrastructure (Moroney, 2017). At the same time, a growing number of users rely on mobile devices as their primary tool for carrying out everyday business tasks, which means that applications intended for such use need to be easy to use and adapted to the limitations of a small screen and an intermittent network connection (Nielsen & Budiu, 2012).

Subject of the Research

The subject of this bachelor's thesis is the development of the mobile application *ESL Lesson Tracker*, intended for ESL teachers for the unified tracking of students, lessons held, payments and monthly earnings. The application was developed using React Native (Expo) for the mobile interface, with Firebase (Firestore and Authentication) as the cloud backend solution, and with the integration of an external API for real-time currency conversion.

Research Objectives

The main objective of the thesis is the design and implementation of a functional mobile solution that enables an ESL teacher to: keep records of students and their contact details; enter and review lessons held, with automatic price calculation; track debts per student with automatic marking of paid lessons (*FIFO method*); and review earnings statistics through monthly and annual reports, including conversion into different currencies. A specific objective is to demonstrate the integration of an external API in real time and the application of the principle of reactive data updating (real-time synchronisation) in a mobile application.

Significance of the Thesis

The significance of the thesis lies in the twofold nature of the results achieved. From a practical point of view, the solution developed is a functional application that is actually used in the everyday work of an ESL teacher, so the thesis goes beyond the scope of an academic exercise and represents a concrete contribution to solving a real problem. The practical value of the solution has also been confirmed through the author's own use of the application, the author being an ESL teacher himself, for whose everyday needs the application was originally developed; according to the author's estimate, regular use of the application instead of the previous manual record-keeping saves approximately four to five hours per week. Informal feedback from several other ESL teachers, to whom the application was demonstrated during the preparation of the thesis, further indicates that the need for such a solution is not limited to an individual case, but is recognised among a wider circle of teachers in the same situation. From a professional point of view, the thesis documents the entire process of developing a mobile application based on a modern technology stack (React Native, Expo, Firebase): from requirements analysis, through the design of the architecture and data model, to implementation and testing, and can therefore serve as useful reference material for students encountering the development of mobile applications built on BaaS platforms for the first time.

Research Methods

The following methods were used in preparing the thesis: the case study method, through the development of a concrete software solution from requirements analysis to implementation and testing; the analytical-synthetic method, in breaking the problem down into functional modules (students, lessons, payments, statistics) and integrating them into a coherent system; and the method of content analysis of secondary sources, through the study

of the technical documentation of the technologies used (React Native, Expo, Firebase). In addition, in preparing the chapter describing the implementation of the solution (Chapter 3), the method of analysing the application's source code was used, which ensures that the functionality and mechanisms described in the thesis correspond exactly to the actual behaviour of the system developed, rather than to a general or idealised description of planned functionality.

Structure of the Thesis

The thesis is organised into five chapters. The first chapter provides an overview of the technologies used. The second chapter covers requirements analysis and system design. The third chapter describes the implementation of the application module by module. The fourth chapter is devoted to testing the solution developed. The concluding remarks summarise the results of the thesis and give recommendations for further improvements to the application. To make the solutions described easier to follow and understand more precisely, the text of the third and fourth chapters is accompanied by the relevant source code excerpts, tables with data structures and test cases, as well as screenshots of the application itself, which directly links the functionality described to its actual appearance and behaviour in use. In this way, the reader gains insight not only into the theoretical description of the solution, but also into its concrete, verifiable realisation.

1. Overview of Technologies

1.1. React Native and Expo

React Native is an open-source framework developed by Meta (formerly Facebook) that enables the development of mobile applications for the iOS and Android platforms using a single JavaScript/TypeScript codebase (Eisenman, 2017; React Native, 2026). Unlike the classical approach to native development, where the application is written separately for each platform (Swift/Objective-C for iOS, Kotlin/Java for Android) (Živković, 2024), React Native translates components written in JavaScript into native UI elements, thereby achieving speed of development while retaining performance close to that of native applications (Boduch & Derks, 2024).

Expo is a set of tools and services built on top of React Native that further simplifies the development process (Expo, 2026). Expo provides the so-called managed workflow, within which the developer does not have to configure native Xcode/Android Studio projects manually; instead, the application is developed, tested and run directly through the Expo Go app or a development build. In this thesis, Expo was used to initialise the project, manage dependencies and run the application during development.

1.2. TypeScript

TypeScript is a superset of the JavaScript programming language that introduces static typing (Boduch & Derks, 2024). Unlike JavaScript, where the type of a variable is determined only at run time, TypeScript requires types to be defined while the code is being written, so that a large number of errors are detected before the application is run, rather than only during testing. In the ESL Lesson Tracker application, TypeScript was used to define the data types corresponding to the Firestore collections (Student, Lesson, Payment), as well as to type the parameters passed between screens via the React Navigation library, which reduces the possibility of errors when accessing non-existent object fields.

1.3. Firebase, Authentication and Firestore

Firebase is a Google platform that belongs to the category of BaaS (Backend as a Service) solutions (Moroney, 2017). The BaaS approach means that the developer does not have to develop and maintain the server infrastructure, database, authentication system or hosting themselves, but instead uses these functionalities as a ready-made service, available through an SDK and API calls.

Firebase Authentication was used to authenticate the application's users by email address and password (Google LLC, 2026). The service takes care of storing credentials securely and generating access tokens, while the application, using the AsyncStorage mechanism, stores the user's session locally on the device, so that the user does not have to sign in again every time the application is launched.

Cloud Firestore is a NoSQL document database that organises data into collections (e.g. students, lessons, payments) and documents within them, unlike relational databases, which use tables and rows (Moroney, 2017). Two approaches to reading data from Firestore were used in this work: real-time onSnapshot listeners, which automatically update the user interface as soon as the data change on the server (e.g. on the Student Detail screen), and one-off getDocs calls, used where real-time updating is not necessary (e.g. when aggregating data for the Dashboard).

1.4. React Navigation

React Navigation is a library that enables navigation to be implemented within a React Native application (React Navigation, 2026). Two types of navigator were used in this work: the Stack Navigator, which organises screens as a stack, with the ability to return to the previous screen, and the Bottom Tab Navigator, which allows the user to move between the main sections of the application (Students, Lessons, Payments, Dashboard) using a bar of icons at the bottom of the screen. Each of the tabs contains its own Stack Navigator, which enables deeper navigation, for example from the list of students to the details of an individual student, without losing the context of the currently selected tab. Data are passed between screens by means of navigation parameters: when moving from the list of students to the details screen, the student's identifier (studentId) is passed, while when moving to the screen for editing data, the entire student object is passed.

1.5. External API, ExchangeRate-API

In order to display earnings in different currencies, the application uses the external service ExchangeRate-API (ExchangeRate-API, 2026). The service is called through an HTTP GET request to the endpoint `https://v6.exchangerate-api.com/v6/{API_KEY}/latest/USD`, which returns the current exchange rates for a large number of world currencies against the US dollar. The call is executed asynchronously every time the Dashboard screen is opened, with a loading

indicator displayed until the response is received. The rates obtained (RSD, HKD, EUR) are used to convert the teacher's total earnings from the base currency into the selected display currency. The service's free plan allows 1,500 calls per month, which is sufficient for the needs of this application given the frequency of calls.

1.6. Other Libraries Used

In addition to the technologies mentioned, a number of additional libraries from the wider React/React Native ecosystem were used in developing the application (Banks & Porcello, 2020). To display statistical data on the Dashboard screen (a bar chart of earnings by month), the react-native-gifted-charts library was used, together with the accompanying dependencies react-native-svg and react-native-linear-gradient, which are required for rendering graphical elements. For date input (lesson date, payment date), the @react-native-community/datetimepicker library was used, which displays a modal spinner on iOS devices and a native date picker dialogue on Android devices. The icons in the navigation bar at the bottom of the screen are provided by the @expo/vector-icons library (the Ionicons icon set).

2. System Analysis and Design

2.1. Requirements Analysis

The requirements analysis considered the functional and non-functional requirements of the LessonLynx application, a tool for tracking individual foreign language tuition intended for the teacher as the sole user of the system. The requirements were derived from the actual implementation of the application and are organised according to the functional modules that correspond to the main navigation sections of the application: Students, Lessons, Payments and Dashboard.

Functional requirements

Students Module

- Creating, editing and deleting student records, storing the basic data: name, optional contact details (email, WeChat ID), the currency in which the student is charged (USD, HKD, EUR or RSD) and the hourly rate (hourlyRate).
- Displaying a list of all students, sorted by name.
- Displaying the details of an individual student, including the current amount owed (totalOwed).
- When a new student is created, the system automatically initialises totalOwed to zero; this field is not subsequently entered manually, but is updated exclusively through operations on lessons and payments (see 2.3).

Lessons Module

- Recording a lesson held: the student to whom the lesson was given, the date, the duration in minutes, the price, the currency, an optional note and the payment status (isPaid).
- Editing and deleting an existing lesson, with automatic adjustment of the student's debt (see 2.3).
- Displaying all lessons (chronologically, from newest to oldest) or lessons filtered by an individual student.

Payments Module

- Recording a payment: student, amount, currency and date.
- When a payment is entered, the system automatically reduces the student's debt (totalOwed) by the amount of the payment and, according to the FIFO principle (first in, first out), marks the student's oldest unpaid lessons as paid (isPaid = true) until the amount of the payment is exhausted.
- Displaying a chronological list of all payments.

Dashboard Module (overview/statistics)

- Displaying aggregated financial indicators by month, in the form of a chart (BarChart), for the selected calendar year.
- Displaying the distribution of earnings by student for the current month (if the current year is selected) or for the whole of the selected year (otherwise), sorted in descending order.
- Displaying the number of active students (students with at least one lesson) in the current month compared with the previous month.
- Converting all amounts into a single reference currency (USD) for comparability and aggregation, using the current exchange rate retrieved from the external ExchangeRate-API service.

Non-functional requirements

- **Real-time data synchronisation:** data on students, lessons and payments are displayed using Firestore real-time listeners (onSnapshot), so that every change to the data (local or from another device) is reflected in the user interface without the need for manual refreshing.
- **Data isolation per user:** all data (students, lessons, payments) are stored as subcollections of the specific user's document (users/{uid}/...), which ensures at the level of the database structure that each user has access exclusively to their own data.

Data isolation per user is further confirmed by the existence of Firestore Security Rules, configured at the level of the Firebase Console (outside the client application's git repository, and therefore not visible in a code search): the rule `match /users/{uid}/{document=**} { allow read, write: if request.auth != null && request.auth.uid == uid; }` explicitly restricts reading and writing on all subcollections (students, lessons, payments) exclusively to the authenticated user whose uid matches the document path.

- **Persistent user session:** authentication is implemented using Firebase Auth, with explicit configuration of session persistence via AsyncStorage (initializeAuth with getReactNativePersistence), so that the user remains signed in between launches of the application, without the need to re-enter their credentials.
- **Resilience to unavailability of the external service:** if the exchange rate service (ExchangeRate-API) is unavailable, the application detects this and does not stop working (the ratesAvailable state), but instead informs the user that the aggregated financial data cannot be displayed.

2.2. Application Architecture

The application's architecture is based on a client-service model, in which the mobile application (the React Native/Expo client) communicates directly with two external systems: the Firebase platform (Authentication and Firestore) and the external REST service ExchangeRate-API. The application has no backend server of its own; all the business logic that would traditionally reside on the server side (validation, aggregation, updating of dependent fields) is implemented on the client, within the service layer.

High-level data flow

1. The user (the teacher) authenticates through Firebase Authentication (email/password); the session token is persisted locally via AsyncStorage.
2. After successful authentication, the client establishes real-time connections (onSnapshot subscriptions) to the Firestore subcollections users/{uid}/students, users/{uid}/lessons and users/{uid}/payments.
3. Every change to the data (whether initiated locally through the UI or by Firestore as confirmation of a write) is automatically propagated through the subscriptions back to the user interface components, without explicit manual refreshing.
4. For the Dashboard module, the client additionally sends an HTTP request to ExchangeRate-API (GET .../latest/USD) to obtain the current conversion rates, which are used exclusively locally, at the time of calculation, and are not persisted in Firestore.

Layers in the code

The application's code is organised in the src/ directory into the following layers, with a clear division of responsibilities:

- **types/ (src/types/index.ts):** a centralised definition of the domain types (Student, Lesson, Payment) and of auxiliary types for display on the Dashboard (MonthData, StudentEarning, ActiveStudentCounts), which constitute the single source of truth for the shape of data throughout the application.
- **services/:** the data access and business logic layer, divided by domain (firebase.ts, students.ts, lessons.ts, payments.ts, dashboard.ts). This layer communicates directly with the Firestore SDK (CRUD operations, real-time subscriptions) and with the external ExchangeRate-API service, and contains logic that goes beyond simple persistence (e.g. the denormalisation of totalOwed, the FIFO allocation of payments, the aggregation of data for charts).
- **hooks/ (useAuthUser.ts):** encapsulates access to the user's Firebase Auth state through a React hook, which the screens components use to obtain the identifier of the currently signed-in user (uid), required for all calls to the services layer.
- **navigation/ (index.tsx):** defines the navigation structure and the typed route parameters (see 2.4), and contains the logic for switching between the unauthenticated and authenticated parts of the application.

- **screens/:** the presentation layer, organised by the same domain units as services (auth, students, lessons, payments, dashboard). Screens components call functions from the services layer, hold local UI state (React useState/useEffect) and render the user interface.
- **components/:** shared presentation components that do not belong to a single screen (e.g. StudentEarningsChart).
- **constants/:** static values used in several places (e.g. month names in constants/months.ts).

This organisation corresponds to a layered architecture in which data access is strictly separated from presentation: screens components do not communicate directly with the Firestore SDK, but exclusively through functions exposed by the services layer, which makes maintenance easier and allows the business logic to be tested in isolation from the user interface.

2.3. Data Model

The application's data model is defined in the file `src/types/index.ts` and is implemented in Firestore as three subcollections within the user's document: `users/{uid}/students`, `users/{uid}/lessons` and `users/{uid}/payments`. This structure (subcollections under the user's document) simultaneously defines both the data model and the per-user data isolation mechanism described in 2.1.

Table 1: Structure of the students collection

Field	Type	Description
id	string	document identifier (assigned by Firestore)
name	string	student's name
email	string (optional)	contact email
wechatId	string (optional)	contact via the WeChat platform
currency	'USD' 'HKD' 'EUR' 'RSD'	currency in which the student is charged
hourlyRate	number	hourly rate
totalOwed	number	current (denormalised) debt of the student
createdAt	Date/Timestamp	time the record was created

Source: author

Table 2: Structure of the lessons collection

Field	Type	Description
id	string	document identifier
studentId	string	reference to the student document
studentName	string	student's name (denormalised)
date	Date/Timestamp	date of the lesson held
durationMinutes	number	duration of the lesson in minutes
price	number	price of the lesson
currency	'USD' 'HKD' 'EUR' 'RSD'	currency of the price
notes	string (optional)	note
isPaid	boolean	whether the lesson has been paid for
createdAt	Date/Timestamp	time the record was created

Source: author

Table 3: Structure of the payments collection

Field	Type	Description
id	string	document identifier
studentId	string	reference to the student document
studentName	string	student's name (denormalised)
amount	number	amount of the payment
currency	'USD'/'HKD'/'EUR'/'RSD'	currency of the payment
date	Date/Timestamp	date of the payment
createdAt	Date/Timestamp	time the record was created

Source: author

Denormalisation of the totalOwed field

The totalOwed field on the student document is a denormalised, precomputed sum, instead of the student's debt being calculated dynamically (by aggregating all of their unpaid lessons) at the time of display. The reason for this approach lies in the limitations of Firestore as a document-oriented database: Firestore does not support real-time SUM-type aggregation queries over a collection on the client side without additionally reading all the relevant documents, which, for every display of the list of students, would require N additional queries (one for each student) on the lessons collection. Instead, totalOwed is maintained incrementally, using Firestore's atomic increment() operation:

- when a lesson is created (addLesson), totalOwed is increased by the price of the lesson (increment(price));
- when a lesson is edited (updateLesson), if the student is the same, totalOwed is adjusted by the difference between the new and old price (increment(newPrice - oldPrice)); if the lesson has been moved to another student, the old price is subtracted from the previous student and the new price is added to the new one;
- when a lesson is deleted (deleteLesson), totalOwed is reduced by the price of the deleted lesson;
- when a payment is entered (addPayment), totalOwed is reduced by the amount of the payment (increment(-amount)).

The increment() operation is atomic at the level of the Firestore server, which avoids the so-called race condition problem that would arise with the classical "read – modify on the client – write" pattern, which is particularly relevant in the context of the real-time synchronisation described in 2.1.

Duplication of the studentName field

The studentName field is duplicated on the documents of the lessons and payments collections (alongside the existing studentId reference), for the same basic reason as the denormalisation of totalOwed: Firestore does not support relational JOIN queries. Without this denormalisation, displaying the list of all lessons or all payments (which includes the student's name next to each item) would require, for each item in the list, an additional read of the corresponding student's document in order to obtain their name, which would significantly increase the number of Firestore reads (and, consequently, costs and latency), especially bearing in mind that the lists of lessons and payments are refreshed in real time through onSnapshot subscriptions. The price of this optimisation is potential inconsistency: if the user later changes a student's name, lessons and payments already recorded retain the old name, because studentName is not updated retroactively on the related documents.

2.4. Navigation Flow and UI/UX Design

Navigation structure

Navigation is implemented using the React Navigation library (@react-navigation/native, version 7) and is centralised in the file src/navigation/index.tsx. The root Navigation component tracks the user's authentication state through the Firebase Auth callback onAuthStateChanged and, on that basis, conditionally renders one of two completely separate navigation units:

- **Auth Stack (AuthStack, createNativeStackNavigator):** displayed when the user is not signed in. It contains two screens, Login and Register, with no header displayed (headerShown: false).
- **Main Tabs (MainTabs, createBottomTabNavigator):** displayed when the user is signed in. It contains four tabs at the bottom of the screen that correspond to the modules described in 2.1: StudentsTab, LessonsTab, PaymentsTab and DashboardTab.

The appearance of the bar with four tabs at the bottom of the screen, with the active tab visually highlighted (primary colour #FFCC99), is shown in Figure 1.

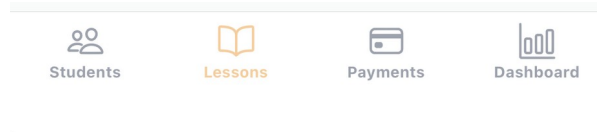


Figure 1 - Tab bar (Bottom Tab Navigator)

Each of the four tabs contains not a single screen but its own nested stack navigator (e.g. StudentsNavigator, LessonsNavigator), which enables each module to have its own navigation depth (e.g. list → details → edit form) independently of the other tabs, while preserving a separate navigation history for each tab. For example, StudentsNavigator contains the screens StudentsList → StudentDetail → AddEditStudent. Signing the currently signed-in user in and out (signOut) is done from the header of the Dashboard screen (a button in the headerRight option of DashboardNavigator), rather than through a separate menu item.

Passing parameters through navigation

Route parameters are typed with TypeScript types defined alongside each stack navigator in navigation/index.tsx (e.g. StudentsStackParamList, LessonsStackParamList, PaymentsStackParamList), so that the compiler checks that every navigation.navigate() call passes the correct shape of parameters for the target screen. Concrete examples of passing data through route parameters:

- **StudentDetail:** receives only { `studentId: string` }, on the basis of which the screen independently loads (subscribes to) the full student object through the services layer;
- **AddEditStudent and AddEditLesson:** receive an optional full object ({ `student?: Student` } and { `lesson?: Lesson` } respectively); the presence of the object signals edit mode, and its absence (undefined) signals the mode for creating a new record, which avoids duplicating the add and edit form in two separate screens;
- **AddPayment:** receives a combination of identifiers and values already known from the previous screen ({ `studentId`, `studentName`, `totalOwed`, `currency` }), so that the data needed to display the context (e.g. the current debt and currency) are passed immediately to the payment entry screen, without an additional read from Firestore when the screen is opened.

Design system

A review of the styles in the screen components showed that the application does not use a centralised design token/theme file (no separate `theme.ts`, `colors.ts` or similar was found); colours and typography are defined locally, within the `StyleSheet.create` call of each individual screen. Despite the absence of centralisation, an analysis of the use of hexadecimal colour values throughout the code shows consistent repetition of the same limited palette, from which an implicit design system can be reconstructed:

- **primary accent colour:** #FFCC99 (pale orange/pastel), used for the active tabs in the bottom navigation and for primary actions;
- **text and headings:** #111827 (almost black, the most frequently used colour in the code);
- **secondary/auxiliary text and inactive icons:** #6B7280 and #9CA3AF (shades of grey);
- **backgrounds and separators:** #F9FAFB, #F3F4F6, #E5E7EB (light shades of grey);
- **semantic state colours:** #EF4444 (red, for errors/deletion/unpaid) and #22C55E (green, for success/paid).

Typography relies on the platform's default system font (no import of a custom font or `fontFamily` property was found in the code), while hierarchy is achieved by varying the size and weight of the type: body text and field labels mostly use a `fontSize` in the range 12–16, and headings and prominent numerical values (e.g. amounts on the Dashboard) a `fontSize` in the range 24–32, with `fontWeight` '600' or '700' for emphasis.

3. Implementation

3.1. Authentication

Firebase initialisation is located in the file `src/services/firebase.ts`. The application uses `initializeApp` to connect to the Firebase project (the configuration is loaded from environment variables with the `EXPO_PUBLIC_` prefix), while Firebase Auth is explicitly initialised through `initializeAuth` with the parameter `persistence: getReactNativePersistence(ReactNativeAsyncStorage)`. In this way, the user's session is stored persistently in `AsyncStorage` on the device, which ensures that the user remains signed in between successive launches of the application, without the need to re-enter their credentials at every launch.

Listing 1: Initialisation of Firebase Auth with AsyncStorage persistence (src/services/firebase.ts)

```
export const auth = initializeAuth(app, {
  persistence: getReactNativePersistence(ReactNativeAsyncStorage),
});
export const db = getFirestore(app);
```

The registration flow is implemented in `src/screens/auth/RegisterScreen.tsx`. Before calling Firebase, the form performs three independent checks on the client: that all fields have been filled in, that the password and the password confirmation match, and that the password is at least 6 characters long. Only after these checks have been passed is `createUserWithEmailAndPassword(auth, email, password)` called. In the event of an error on the Firebase side, the user is shown a readable message mapped through the same error dictionary that is used by the sign-in screen (see below).

The sign-in flow is implemented in `src/screens/auth/LoginScreen.tsx`, using `signInWithEmailAndPassword(auth, email, password)`. The sign-in screen uses an error dictionary extracted into the shared module `src/constants/authErrors.ts` (`AUTH_ERROR_MESSAGES`, `AUTH_ERROR_FALLBACK`), following the already existing pattern of shared constants (`constants/months.ts`). The same dictionary is also used by the registration screen, which removes the earlier asymmetry in error handling between the two screens:

Listing 2: Shared dictionary for mapping Firebase errors (src/constants/authErrors.ts)

```
export const AUTH_ERROR_MESSAGES: Record<string, string> = {
  'auth/invalid-credential': 'Incorrect email or password.',
  'auth/user-not-found': 'No account found with this email.',
```

```

    'auth/wrong-password': 'Incorrect password.',
    'auth/invalid-email': 'Please enter a valid email address.',
    'auth/too-many-requests': 'Too many failed attempts. Try again later.',
    'auth/email-already-in-use': 'An account with this email already
exists.',
    'auth/weak-password': 'Password must be at least 6 characters.',
    'auth/operation-not-allowed': 'Registration is currently disabled.
Please try again later.',
  };
export const AUTH_ERROR_FALLBACK = 'Something went wrong. Please try
again.';

```

Both screens call `AUTH_ERROR_MESSAGES[e.code]` ?? `AUTH_ERROR_FALLBACK` when displaying the Alert dialogue. If the error code returned by Firebase is not explicitly listed in the dictionary, the user is shown a generic fallback message instead of a technical description of the error.

The appearance of the sign-in screen is shown in Figure 2, while an example of a mapped error message being displayed (when an email address in an invalid format is entered) is shown in Figure 3.

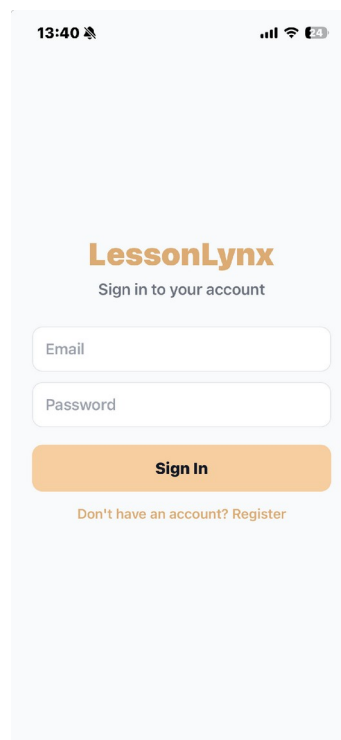


Figure 2 - Sign-in screen (Login)

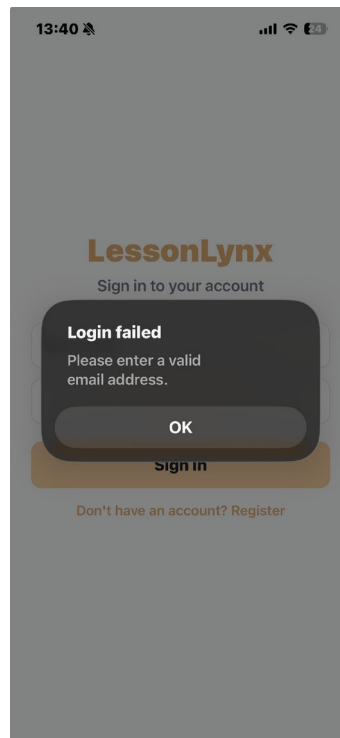


Figure 3 - Mapped error message on sign-in

Switching between Auth and Main navigation is managed in the root Navigation component (`src/navigation/index.tsx`), which holds its own local user and loading state and subscribes to Firebase's `onAuthStateChanged` callback independently of the `useAuthUser` hook:

Listing 3: `onAuthStateChanged` subscription in the root navigation component

```
useEffect(() => {
  const unsub = onAuthStateChanged(auth, (u) => {
    setUser(u);
    setLoading(false);
  });
  return unsub;
}, []);
```

While the initial check of the authentication state is in progress (`loading === true`), an `ActivityIndicator` is displayed. Once the callback returns a result for the first time, the component conditionally renders `<MainTabs />` if user is set, or `<AuthStack />` (with the Login and Register screens) if the user is not signed in. Signing the user out (`signOut(auth)`) is invoked from the header of the Dashboard screen, after confirmation in an Alert dialogue.

A separate hook, `useAuthUser` (`src/hooks/useAuthUser.ts`), encapsulates the same subscription to `onAuthStateChanged`, but is used in the code only within the

DashboardScreen component, to obtain the identifier of the signed-in user (user.uid) needed to load data for the charts.

3.2. Students Module

CRUD operations on students are implemented in `src/services/students.ts` and rely directly on the Firestore Web SDK, without an additional abstraction layer. All operations address the `users/{uid}/students` subcollection. Creating a student (`addStudent`) receives data without the `id`, `createdAt` and `totalOwed` fields, and explicitly sets `totalOwed` to 0 and `createdAt` via `serverTimestamp()` before writing. Editing (`updateStudent`) passes only the changed fields to `updateDoc`. The form for adding/editing a student, with a choice of one of the four supported currencies, is shown in Figure 8.

The screenshot shows a mobile application interface for adding a new student. At the top, the status bar displays the time 14:18 and battery level 29%. Below the status bar, there's a navigation bar with a back arrow and the text 'Students', and a 'New Student' button. The main form area has a light gray background and contains several input fields: 'NAME *' with the value 'Nathan', 'LESSON RATE *' with the value '10', 'DEFAULT CURRENCY *' with four buttons (USD, HKD, EUR, RSD) where EUR is selected, 'EMAIL (OPTIONAL)' with the value 'nathan@example.com', and 'WECHAT ID (OPTIONAL)' with the value 'e.g. john_wechat'. At the bottom of the form is an orange 'Add Student' button. Below the form is a bottom navigation bar with four icons: 'Students' (selected), 'Lessons', 'Payments', and 'Dashboard'.

Figure 8 - Form for adding a student (Add/Edit Student)

Deleting a student (`deleteStudent`) is implemented as an atomic batch operation using Firestore's `writeBatch` mechanism: the function first uses a query (`where('studentId', '==', studentId)`) to find all related documents in the lessons and payments collections, adds their deletion to the batch together with the deletion of the student's own document, and executes all the changes with a single `batch.commit()` call. This prevents "orphaned" records from remaining in the database after a student is deleted, and the operation, unlike the other CRUD operations described in this chapter, is performed as all-or-nothing (atomic), with no risk of partial execution in the event of an error between individual calls.

`StudentsListScreen` (`src/screens/students/StudentsListScreen.tsx`) displays the list of all students using the real-time subscription `subscribeToStudents`. Search is implemented

entirely on the client side, as filtering of the array already loaded into memory, without additional Firestore queries:

Listing 4: Client-side filtering of the student search

```
const filtered = students.filter((s) =>
  s.name.toLowerCase().includes(search.toLowerCase())
);
```

The list is not sorted in the component itself; sorting has already been applied at the level of the Firestore query in the service layer: `subscribeToStudents` uses `query(..., orderBy('name'))`, so the client always receives data already sorted by name. The appearance of the list of students, with students who owe money (in red, with the amount) visually distinguished from those who have settled their debt (in green, "Paid up"), is shown in Figure 4.

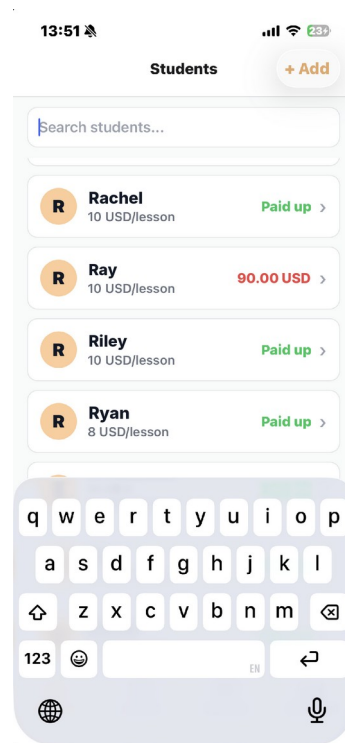


Figure 4 - List of students (StudentsList) with debt statuses

`StudentDetailScreen` (`src/screens/students/StudentDetailScreen.tsx`) uses two independent real-time subscriptions at the same time: `subscribeToStudent` (for the basic data and `totalOwed` of the specific student) and `subscribeToStudentLessons` (for all of that student's lessons). Both subscriptions are cancelled (`unsubStudent()`, `unsubLessons()`) when the component unmounts, in the cleanup function of the `useEffect` hook. The lessons displayed are divided into two sections on the basis of the `isPaid` field:

Listing 5: Dividing lessons into unpaid and paid

```
const unpaidLessons = lessons.filter((l) => !l.isPaid);  
const paidLessons = lessons.filter((l) => l.isPaid);
```

The section of unpaid lessons is always displayed expanded (if there is at least one unpaid lesson), while the section of paid lessons is collapsed by default and can be expanded/collapsed through the local UI state `paidExpanded`, so that the user is shown the data relevant to the current debt first. An example of a student with a debt, with the section of unpaid lessons expanded, is shown in Figure 5. An example of a student with no debt, with the section of paid lessons collapsed (Figure 6) and expanded (Figure 7), in which each paid lesson carries a green confirmation mark, is shown in the following figures.

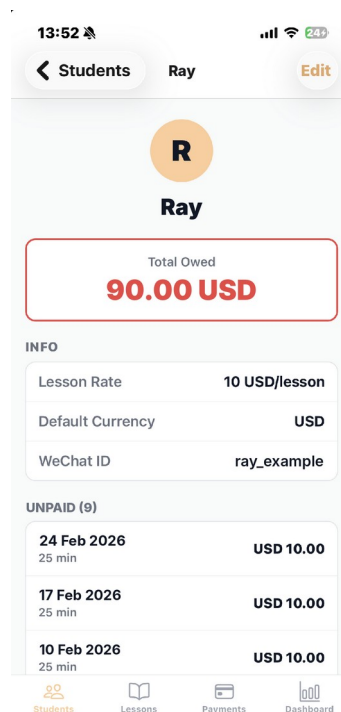


Figure 5 - Student detail with unpaid lessons (StudentDetail)

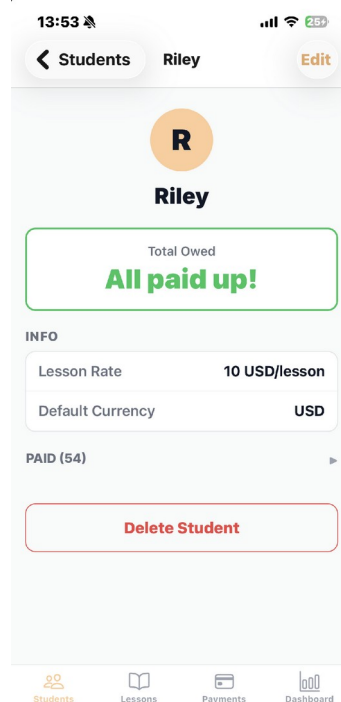


Figure 6 - Student detail, paid lessons section collapsed

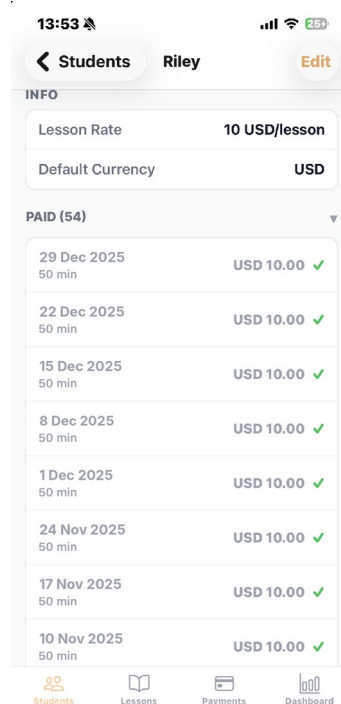


Figure 7 - Student detail, paid lessons section expanded

3.3. Lessons Module

The form for adding and editing a lesson is located in `src/screens/lessons/AddEditLessonScreen.tsx`. The duration of a lesson is restricted to three predefined values (25, 50 and 75 minutes), each of which corresponds to a fixed, predefined price through a constant mapping:

Listing 6: Fixed prices by lesson duration

```
const DURATIONS = [25, 50, 75];  
const DURATION_PRICES: Record<number, number> = { 25: 10, 50: 20, 75:  
30 };
```

In the mode for adding a new lesson (`isEdit === false`), the price field is automatically set to the corresponding value from `DURATION_PRICES` every time the user changes the selected duration, through a separate `useEffect` effect. The price field (`TextInput`) remains editable at all times, so the user can manually overwrite the automatically suggested value (override); this automatic synchronisation is, however, disabled in the mode for editing an existing lesson (`isEdit === true`), where the originally entered price is retained and is not automatically overwritten when the duration changes. An example of the lesson edit screen is shown in Figure 9: a 50-minute lesson retains a price of 10 (the originally entered value) instead of being automatically set to 20, precisely because of this behaviour.

The screenshot shows a mobile application interface for editing a lesson. At the top, there's a status bar with the time 14:57 and signal indicators. Below it, a navigation bar has a back arrow and the text 'Lessons Edit Lesson'. The main form area is titled 'STUDENT' and contains a dropdown menu with 'Brooke' selected. Below that is a 'DATE' section with a dropdown menu showing '20 Nov 2025'. The 'DURATION' section has three buttons: '25 min', '50 min' (which is highlighted in orange), and '75 min'. The 'PRICE' section has a text input field containing the number '10'. The 'CURRENCY' section has four buttons: 'USD' (highlighted in orange), 'HKD', 'EUR', and 'RSD'. At the bottom of the form is a 'NOTES (OPTIONAL)' section with a text area containing the placeholder text 'Any notes about this lesson...'. The bottom of the screen features a navigation bar with four icons: 'Students', 'Lessons' (highlighted in orange), 'Payments', and 'Dashboard'.

Figure 9 - Lesson edit form (Edit Lesson), with a choice of duration and price

When saving, the currency of the lesson is taken from the selected student at the moment the student is chosen from the list (`handleSelectStudent` sets `setCurrency(student.currency)`), but it can subsequently be changed using separate currency buttons.

`LessonsListScreen` (`src/screens/lessons/LessonsListScreen.tsx`) does not display a flat list of lessons; instead, it groups them by student using the `useMemo` hook, forming a map by `studentId`, and then sorts the resulting sections by student name (`localeCompare`). Each section can be expanded/collapsed individually, through a local `Set` called `expanded`, which stores the identifiers of the currently open sections. The list with several collapsed sections per student is shown in Figure 10, while an example of one expanded section, with individual lessons, is shown in Figure 11.

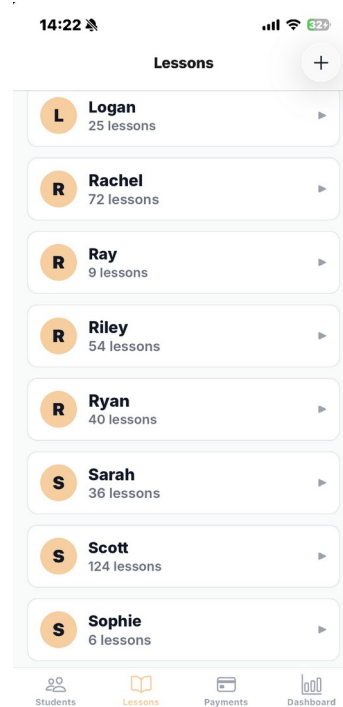


Figure 10 - List of lessons grouped by student (collapsed sections)

Lessons	
Hailey 47 lessons	
5 Jan 2026 50 min	USD 10.00
2 Jan 2026 50 min	USD 10.00
29 Dec 2025 50 min	USD 10.00
22 Dec 2025 50 min	USD 10.00
19 Dec 2025 50 min	USD 10.00
15 Dec 2025 50 min	USD 10.00
12 Dec 2025 50 min	USD 10.00
8 Dec 2025 50 min	USD 10.00
5 Dec 2025 50 min	USD 10.00
27 Nov 2025	

Figure 11 - List of lessons, one student's section expanded

The mechanism for updating `totalOwed` is implemented in `src/services/lessons.ts` and is the concrete realisation of the denormalisation described in Section 2.3. When a lesson is created, after the document has been written to the `lessons` collection, a separate, sequential call is made that increases the student's `totalOwed` using the `atomic increment()` operation:

Listing 7: Updating `totalOwed` when a lesson is created (`addLesson`)

```
export async function addLesson(uid, data) {
  await addDoc(lessonsCol(uid), { ...data, createdAt:
serverTimestamp() });
  await updateDoc(studentDoc(uid, data.studentId), {
    totalOwed: increment(data.price),
  });
}
```

When a lesson is edited (`updateLesson`), the logic depends on whether the lesson has been moved to another student. If the student remains the same, `totalOwed` is adjusted only by the difference between the new and old price; if the student has changed, the price is first subtracted from the old student and then added to the new one:

Listing 8: Updating `totalOwed` when a lesson is edited (`updateLesson`)

```
if (oldLesson.studentId === data.studentId) {
  const diff = data.price - oldLesson.price;
  if (diff !== 0) {
```

```
    await updateDoc(studentDoc(uid, data.studentId), { totalOwed:
increment(diff) });
  }
} else {
  await updateDoc(studentDoc(uid, oldLesson.studentId), { totalOwed:
increment(-oldLesson.price) });
  await updateDoc(studentDoc(uid, data.studentId), { totalOwed:
increment(data.price) });
}
```

When a lesson is deleted (`deleteLesson`), the document is first deleted from the lessons collection, and the student's `totalOwed` is then reduced by the price of the deleted lesson (`increment(-lesson.price)`). In all three cases, the operations on the lesson document and on the student document are performed as two separate, sequential async calls, rather than within a Firestore transaction (`runTransaction`) or a batched write (`writeBatch`).

3.4. Payments Module and FIFO Logic

The Payments screen displays a list of all students, with students who owe money singled out in a "Payment due" section at the top of the screen (in red), while students with no debt are shown below (in green, "Paid up"), as in Figure 12. Tapping a student with a debt opens the payment entry screen (Figure 13), on which the amount is prefilled with the student's current debt.

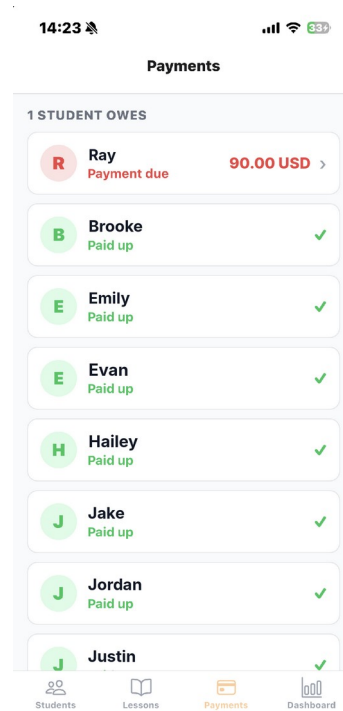


Figure 12 - Payments screen with students who owe money and those who have settled their debt

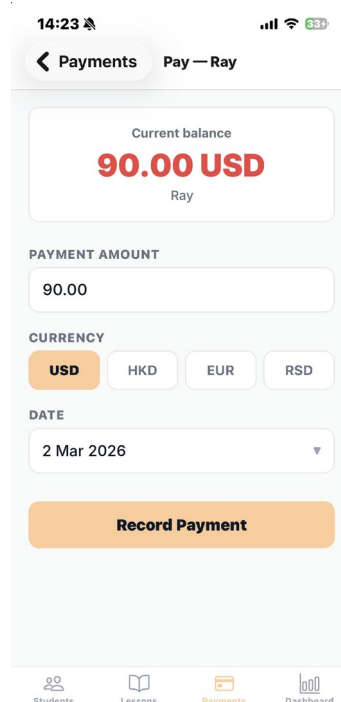


Figure 13 - Payment entry screen (Add Payment)

The `addPayment` function in `src/services/payments.ts` performs three operations in succession each time a payment is entered. The first is a simple write of the document to the `payments` collection. The second is the reduction of the student's `totalOwed` via `increment(-data.amount)`, analogous to the mechanism described in 3.3. The third, and the most complex, is the allocation of the payment to individual lessons according to the FIFO principle (first in, first out): the oldest unpaid lessons are paid off first.

The algorithm is executed in four steps:

1. Retrieving all of the student's lessons. Instead of filtering by `isPaid` at the level of the Firestore query, all lessons of the given student are retrieved (`where('studentId', '==', data.studentId)`), and filtering by payment status is performed afterwards, on the client.
2. Filtering and sorting unpaid lessons. From the documents retrieved, only those with `isPaid === false` are selected, and they are then sorted in ascending order by date (oldest first).

Listing 9: Filtering and sorting unpaid lessons (FIFO preparation)

```
const unpaid = lessonsSnap.docs
  .filter((d) => !d.data().isPaid)
  .map((d) => ({ ref: d.ref, price: d.data().price, date:
(d.data().date).toDate() })))
  .sort((a, b) => a.date.getTime() - b.date.getTime());
```

3. Iteratively consuming the amount of the payment. The sorted list is traversed in order, and the variable remaining tracks the remaining, not yet allocated amount of the payment. A lesson is marked as paid (isPaid: true) only if the remaining amount fully covers its price.

Listing 10: Iteratively marking lessons as paid (FIFO loop)

```
let remaining = data.amount;
for (const lesson of unpaid) {
  if (remaining <= 0) break;
  if (remaining >= lesson.price) {
    await updateDoc(lesson.ref, { isPaid: true });
    remaining -= lesson.price;
  }
}
```

4. Terminating the loop. The loop terminates as soon as remaining falls to 0 or below, or as soon as the list of unpaid lessons is exhausted.

Edge cases

If remaining is less than the price of the current lesson (lesson.price), the branch if (remaining >= lesson.price) is not executed, the lesson is not marked as paid, and the loop continues to the next lesson in the sequence instead of stopping, which means that a payment that partially covers one lesson does not "skip ahead" to a smaller later lesson that it could cover in full; instead, the iteration proceeds strictly in chronological order, without any attempt at optimal matching. If the amount of the payment is greater than the total of all unpaid lessons (e.g. the user pays in advance or overpays), all unpaid lessons are marked as isPaid: true, and the student's totalOwed (already reduced by the full amount of the payment) becomes a negative number. Unlike in an earlier version of the application, this surplus is now explicitly shown to the user: StudentsListScreen, StudentDetailScreen and PaymentsScreen check, in addition to the existing condition totalOwed > 0 ("owes"), the condition totalOwed < 0 ("hasCredit"), in which case "Credit: X" with the absolute value of the overpaid amount is displayed instead of the "Paid up" label. The grouping of students (owing/paidUp) remains unchanged, because a student in credit formally owes nothing; only the visual display of the individual item changes.

Furthermore, since the operations within the for loop (updateDoc(lesson.ref, ...)) are not grouped in a writeBatch, in the event of a network error in the middle of the iteration, some lessons may remain marked as paid and others not, with no possibility of automatically reverting to the previous state (rollback).

3.5. Dashboard and External API

The service layer of the Dashboard module (`src/services/dashboard.ts`) separates the retrieval of raw data from their aggregation. The `fetchAllLessons` function retrieves all of the user's lessons from Firestore (once, via `getDocs`, rather than as a real-time subscription), while `fetchExchangeRates` calls the external REST service `ExchangeRate-API`:

Listing 11: Calling the external API for exchange rates (`fetchExchangeRates`)

```
export async function fetchExchangeRates(): Promise<Record<string,
number> | null> {
  try {
    const apiKey = process.env.EXPO_PUBLIC_EXCHANGERATE_API_KEY;
    const res = await fetch(`https://v6.exchangerate-api.com/v6/${
      apiKey}/latest/USD`);
    const json = await res.json();
    if (json.result === 'success') return json.conversion_rates as
Record<string, number>;
    return null;
  } catch {
    return null;
  }
}
```

Both calls (`fetchAllLessons` and `fetchExchangeRates`) are executed in parallel through `Promise.all` within the `DashboardScreen` component, on initial loading and when the signed-in user changes. If `fetchExchangeRates` returns null (a network error, an unsuccessful HTTP response or `json.result !== 'success'`), the component sets the local state `ratesAvailable` to false and stops any further calculation of aggregates.

Aggregation by month is performed by the `buildAnnualData` function, which generates an array of exactly 12 elements for the selected year and, for each month, calculates the sum of lesson prices (converted to USD via `convertToUSD`) and the number of lessons held in that month. For the current year, months that are yet to come are marked with the field `isFuture: true` and their sum is not calculated, while the current month is specially marked with the field `isCurrent: true` so that it can be highlighted on the chart.

Aggregation by student is performed via `buildStudentMonthlyEarnings` (for the current year, grouping by the current month) or `buildStudentAnnualEarnings` (for previous years, grouping by the whole year); `DashboardScreen` selects one of these two functions depending on whether `selectedYear` is equal to the current year. Both functions group lessons by `studentId`, sum the converted amounts and calculate each student's percentage share of the total (percentage).

The chart is rendered using the BarChart component from the react-native-gifted-charts library. The chart data are formed by mapping the output of the buildAnnualData function, with the colour of each bar determined conditionally: the selected (tapped) bar is coloured #F59E0B, the current month #FFCC99, future months (without data) #F3F4F6, and past months #E5C08A. Tapping a bar (onPress) sets the local state selectedBarIdx, which then controls both the colour and the textual display of the details above the chart. The width of the chart is calculated dynamically on the basis of the number of bars, so that a chart with several months of data becomes horizontally scrollable instead of the bars being squeezed below the width of the screen. When ratesAvailable is false, both the BarChart and the StudentEarningsChart components are replaced by a text message saying that the exchange rate is unavailable, instead of attempting to display incomplete or incorrect data. The appearance of the statistics cards and the interactive chart of earnings by month is shown in Figure 14, while the ranking of revenue by student, together with the card showing live exchange rates (ExchangeRate-API), is shown in Figure 15.



Figure 14 - Dashboard: statistics and chart of earnings by month

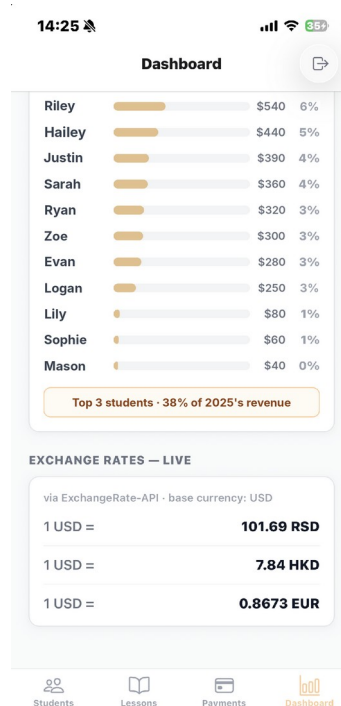


Figure 15 - Dashboard: ranking of revenue by student and live exchange rates

4. Testing

No automated tests were found in the project: there is no `tests/` directory, there are no files with the extension `.test.ts(x)` or `.spec.ts(x)`, there is no configuration file for Jest (`jest.config.*`) or for Detox (`.detoxrc`), and `package.json` defines no test script, nor are the libraries `jest`, `jest-expo`, `detox` or `@testing-library/*` listed among the dependencies. It follows that testing of the application was not automated, but exclusively manual/functional, carried out by using the application directly (Expo Go / simulator) during development. The scenarios described below represent a reconstruction of such manual testing, based on the actual business logic implemented in the service layer (`services/`), rather than on test cases formally defined and executed within a testing framework.

4.1. Functional Testing

Students Module (`services/students.ts`)

Table 4: Test cases: Students module

ID	Scenario	Steps	Expected result
TC-ST-01	Creating a student and the initial value of <code>totalOwed</code>	1. Open <code>StudentsList</code> → "+ Add". 2. Enter a name, rate and currency. 3. Save (<code>addStudent</code>).	The new document appears through the real-time subscription; <code>totalOwed</code> is 0, even though it was not explicitly entered (<code>addStudent</code> sets it explicitly before writing).
TC-ST-02	Editing a student's data	1. Open <code>StudentDetail</code> . 2. "Edit" → change the rate/currency. 3. Save (<code>updateStudent</code>).	The changes are reflected immediately without manual refreshing (<code>onSnapshot</code>). <code>totalOwed</code> remains unchanged.
TC-ST-03	Deleting a student	1. <code>StudentDetail</code> → "Delete Student" → confirm. 2. <code>deleteStudent</code> .	The document is removed from students; the user is returned to <code>StudentsList</code> .
TC-ST-04	Deleting a student also deletes the related records	1. Create a student with at least one lesson and one payment. 2. Delete the student (<code>deleteStudent</code>). 3. Check the lessons and payments collections for that student.	The documents in lessons and payments that reference the deleted student are deleted together with the student, within the same atomic batch operation (<code>writeBatch</code>); it is confirmed that no orphaned records remain in the database.
TC-ST-05	Search on <code>StudentsList</code>	1. Enter part of a name in the search field.	The list is filtered on the client, without an additional Firestore query.

Source: author

Lessons Module (services/lessons.ts)

Table 5: Test cases: Lessons module

ID	Scenario	Steps	Expected result
TC-LS-01	Adding a lesson and increasing totalOwed	1. Note totalOwed before the entry. 2. Add a lesson priced at 20 (addLesson).	totalOwed increases by 20, as a result of increment(price).
TC-LS-02	Changing the price for the same student	1. Change the price from 20 to 30, same student (updateLesson).	totalOwed is adjusted by the difference (+10); it is not set directly to 30.
TC-LS-03	Moving a lesson to another student	1. Change the studentId of an existing lesson from A to B.	totalOwed(A) decreases and totalOwed(B) increases by the same price; the total of the debts remains the same.
TC-LS-04	Deleting a lesson	1. Delete an unpaid lesson (deleteLesson).	totalOwed decreases by exactly the price of the deleted lesson.
TC-LS-05	Automatic price when adding	1. Select a duration of 25, then 50, then 75 min.	The price is automatically set to 10, 20, 30 (DURATION_PRICES).
TC-LS-06	Manual price override	1. Duration 50 min (automatic price 20). 2. Manually enter 25. 3. Save.	The lesson is saved with a price of 25; the field remains editable and overrides the suggestion.
TC-LS-07	Grouping by student on LessonsList	1. Add lessons for two students. 2. Open LessonsList.	Lessons are displayed grouped and sorted by name (localeCompare).

Source: author

Payments Module (services/payments.ts)

Table 6: Test cases: Payments module

ID	Scenario	Steps	Expected result
TC-PM-01	Payment that exactly covers the debt (FIFO)	1. The student has an unpaid lesson of 20. 2. A payment of exactly 20.	totalOwed falls to 0; the lesson isPaid: true (remaining >= price holds exactly at the boundary).
TC-PM-02	Payment that partially covers a lesson (edge case)	1. An unpaid lesson of 30. 2. A payment of 20 (remaining=20 < 30).	totalOwed falls to 10, but the lesson remains isPaid: false; no lesson is marked as paid.
TC-PM-03	Payment that exceeds the total debt (overpayment)	1. The student has unpaid lessons totalling 20 (one lesson). 2. Enter a payment of 50.	All of the student's unpaid lessons are marked isPaid: true. totalOwed falls to -30 (increment(-50) applied to the previous 20).
TC-PM-04	Displaying an overpayment in the user interface	1. Continue from TC-PM-03 (totalOwed = -30). 2. Open StudentsList, StudentDetail and PaymentsScreen.	The student is shown with the label "Credit: 30.00 [currency]" instead of "Paid up": StudentsListScreen shows "Credit: X" in the card's colour, StudentDetailScreen changes the card title from "Total Owed" to "Credit" and displays the absolute value, and PaymentsScreen shows the status "Credit" and the amount instead of a tick (✓). The student still falls into the "paid up" group (owes nothing), but is visually distinguished from a student with totalOwed === 0.
TC-PM-05	Order of payment allocation with several unpaid lessons	1. Older lesson of 10 (1 March), newer lesson of 20 (15 March). 2. A payment of 10.	Only the older lesson is paid off; the list is sorted in ascending order by date.

Source: author

Dashboard (services/dashboard.ts)

Table 7: Test cases: Dashboard

ID	Scenario	Steps	Expected result
TC-DB-01	Aggregation when ExchangeRate-API is available	1. Open the Dashboard with a valid API key and a network connection.	fetchExchangeRates returns conversion_rates; ratesAvailable=true; the charts and exchange rates are displayed.
TC-DB-02	Behaviour when the API is unavailable	1. Simulate unavailability of the service before opening the Dashboard.	ratesAvailable=false; aggregates are not calculated (early return); "unavailable" messages are displayed instead of charts.
TC-DB-03	"This month" versus "Annual total"	1. Current year → check the label. 2. Move to the previous year using the arrow.	For the current year, "This month" is displayed; for the previous year, "Annual total" (the total for the whole year).
TC-DB-04	Active students — comparison of months	1. Note the number of active students in the current and previous months.	A delta (" +N/-N/Same as last month") is displayed in the appropriate colour.

Source: author

4.2. Authentication Testing and Known Issues

Sign-in and session scenarios

Table 8: Test cases: authentication

ID	Scenario	Steps	Expected result
TC-AU-01	Successful sign-in	1. Correct email and password. 2. "Sign In".	onAuthStateChanged sets user; MainTabs is displayed.
TC-AU-02	Wrong password	1. Correct email, wrong password.	Alert "Incorrect password." or "Incorrect email or password."
TC-AU-03	Non-existent account	1. Enter an unregistered email.	Message "No account found with this email."
TC-AU-04	Invalid email format	1. Enter text that is not a valid email address.	Message "Please enter a valid email address."
TC-AU-05	Unmapped error code	1. Trigger an error outside the dictionary (e.g. auth/network-request-failed).	Fallback message "Something went wrong. Please try again."
TC-AU-06	Passwords do not match (registration)	1. Different values in Password/Confirm Password.	Local message "Passwords do not match." before the call to Firebase.
TC-AU-07	Session persistence after a restart	1. Sign in. 2. Force-close the app. 3. Launch it again.	A brief ActivityIndicator, then MainTabs directly, without the Login screen.
TC-AU-08	Sign-out	1. Dashboard → sign-out icon → confirm.	signOut(auth); user becomes null; AuthStack is displayed.
TC-AU-10	Unsuccessful registration, email already in use	1. On the Register screen, enter the email of an existing account.	The message "An account with this email already exists." (auth/email-already-in-use) is displayed, through the same AUTH_ERROR_MESSAGES dictionary that is used by the Login screen.

Source: author

Case study: a bug found and fixed, a blank (white) DateTimePicker spinner on iOS

In the project documentation (PRD.md, section 17.0 "Bug: Date Picker Blank on iOS"), a bug discovered during manual testing of date entry on iOS was formally logged and described. Screen: AddEditLessonScreen (it was subsequently confirmed that the same problem also occurs in AddPaymentScreen). Symptom: on tapping the date field, the modal with the spinner picker (display="spinner") opens correctly, but the content of the spinner itself is blank/invisible, white text on a white background. Cause: on newer versions of iOS, the DateTimePicker component with display="spinner" renders text in white by default; without the themeVariant="light" property set explicitly, the spinner text remains invisible

against the light background of the modal. Solution: adding the `themeVariant="light"` prop to the `DateTimePicker` component within the iOS modal on both affected screens.

The fix is also confirmed in the project's git history: commit `e59c056` ("final bugs and changes completed") contains an identical change in both files:

Listing 12: Fix for the DateTimePicker bug (git diff, commit e59c056)

```
<DateTimePicker
  value={tempDate}
  mode="date"
  display="spinner"
+  themeVariant="light"
  onChange={(_, selected) => {
    if (selected) setTempDate(selected);
  }}
/>
```

A regression test after the fix (TC-AU-09, by analogy with the scenarios listed above) would consist of reopening the date-picker modal on both screens (`AddEditLessonScreen` and `AddPaymentScreen`) on an iOS device/simulator and visually checking that the day, month and year digits in the spinner are legible (dark text on a light background).

Note on the development of sign-in error messages

PRD.md (section 17.3, "Custom Auth Error Messages") planned the introduction of readable error messages on unsuccessful sign-in, with a proposed implementation in the form of a switch statement within the helper function `getAuthErrorMessage`. The checklist items in this section of PRD.md were left unticked (incomplete); however, the actual implementation (described in Section 3.1) achieves a functionally identical, and indeed broader, outcome, mapping an extended set of Firebase error codes shared by the Login and Register screens, but through a static `Record<string, string>` dictionary extracted into a separate module (`constants/authErrors.ts`), rather than through a switch construct within the screen itself.

Concluding Remarks

In this bachelor's thesis, the mobile application LessonLynx (ESL Lesson Tracker) was developed, intended for a self-employed teacher of English as a foreign language, for the unified tracking of students, lessons held, payments and monthly/annual earnings. The application was built using React Native/Expo for the mobile interface, with Firebase (Authentication and Firestore) as the cloud backend solution, and with the integration of the external ExchangeRate-API service for converting earnings into several currencies in real time.

Through requirements analysis, system design, implementation and testing, it has been shown that the solution developed meets the objectives set: it enables the recording of students with basic contact details, the entry and review of lessons with an automatic price suggestion based on duration, the tracking of debts per student with automatic allocation of payments to the oldest unpaid lessons (the FIFO principle), and the review of aggregated earnings statistics through an interactive chart by month and a ranking of revenue by student. Real-time data synchronisation, achieved through Firestore onSnapshot subscriptions, ensures that changes are immediately reflected in the user interface without the need for manual refreshing, while the persistent Firebase Auth session allows the user to work without interruption and without repeatedly signing in.

Achievement of the research objectives

In comparison with the research objectives defined in the introductory part of the thesis, it can be stated that all the objectives set have been fully achieved:

- **Keeping records of students and their contact details:** achieved through the Students module, including the name, contact details, billing currency and hourly rate, with the debt automatically initialised to zero for every new student (Sections 2.1 and 3.2).
- **Entering and reviewing lessons with automatic price calculation:** achieved through the Lessons module, with fixed lesson durations (25/50/75 minutes) to which an automatic price suggestion corresponds, with the option of manually overwriting the suggested value (Section 3.3).
- **Tracking debts with automatic marking of paid lessons:** achieved through the Payments module and the FIFO algorithm implemented in the addPayment function, which pays off the oldest unpaid lessons first, with atomic updating of debts through Firestore's increment() operation (Section 3.4).
- **Reviewing earnings statistics with currency conversion:** achieved through the Dashboard module, which displays monthly and annual earnings, a ranking of

revenue by student and conversion into different currencies using current exchange rates from the ExchangeRate-API service (Section 3.5).

The specific objective, demonstrating the integration of an external API in real time and the application of the principle of reactive data updating, has also been achieved, both through the ExchangeRate-API integration described and through the consistent use of Firestore onSnapshot subscriptions on several screens of the application (Students, Lessons, StudentDetail).

Limitations of the solution

During analysis and implementation, a number of limitations of the current solution were identified, which represent room for improvement before any wider use of the application:

- **Inconsistency with concurrent changes:** the operations that update a student's totalOwed (when adding/editing/deleting a lesson, as well as when entering a payment) are performed as two separate, sequential Firestore calls, rather than within a transaction (runTransaction) or a batched write (writeBatch), which, in the event of a network error mid-execution, could theoretically lead to a mismatch between the totalOwed value and the actual state of the lessons. A partial exception is the operation of deleting a student, which, unlike the others, is already performed as an atomic writeBatch operation (see 3.2).
- **Absence of automated testing:** the application has no automated tests (unit or integration), so the correctness of the business logic relies exclusively on the manual testing described in the fourth chapter. Data isolation per user, on the other hand, is also confirmed at the level of the Firestore Security Rules configured in the Firebase Console (see 2.1).
- **Implicit design system:** the colour palette and typography are not defined in a single centralised place (e.g. theme.ts), but are repeated locally in the styles of each screen, which makes future changes to the application's visual identity more difficult.
- **Outdated student name on old records:** the studentName field on documents in the lessons and payments collections is not updated retroactively if the user later changes a student's name (see 2.3); this is a conscious trade-off of denormalisation, not an oversight.

Reflection on the development process

The process of developing the application confirmed several principles that were presented theoretically in the technology analysis phase (Chapter 1). Data denormalisation (totalOwed, studentName), introduced to adapt to Firestore's query model, proved necessary even with the small number of students and lessons used during testing; without it, every display of a list would have required additional database reads for each item. On the other hand, this very denormalisation gave rise to most of the limitations described in the previous

section (the risk of inconsistency with concurrent changes, outdated student names on old records), which points to a typical trade-off in designing applications based on document-oriented databases: the gain in the number of reads and display speed is paid for with the additional complexity of maintaining consistency. The implementation of the FIFO logic for allocating payments proved to be the most complex single part of the business logic in the thesis, precisely because of the need to correctly handle edge cases (a partially covered lesson, an overpayment) that are not obvious at first glance from the basic description of the requirements, and which were discovered only through detailed analysis of the code and the design of the test scenarios in the fourth chapter.

Possible extensions

On the basis of the limitations identified and the needs that naturally arise from everyday use of the application, the following directions for further development stand out:

- **Weekly timetable view:** introducing a weekly calendar view of lessons (days of the week as columns, time slots as rows) would be a significant improvement on the current list view grouped by student, making it easier to plan and to spot free slots and any overlaps in the timetable.
- **Generating and sending invoices:** for students with a larger number of unpaid lessons, the application could automatically generate an invoice (a statement of debt) in a shareable format, which the teacher could send to the student with a single tap via messaging apps, with the option of marking all of the student's lessons as paid in bulk after the payment has been confirmed.
- **Tracking the cancellation rate per student:** an additional statistic that tracked the ratio of scheduled to cancelled lessons for each student (the cancellation percentage) would help the teacher to identify less reliable students and adjust their priority accordingly when scheduling new slots.
- **Overview of the teacher's workload:** a visual overview (heatmap) of the total number of teaching hours distributed across the days of the week and parts of the day (morning/afternoon/evening) would give the teacher better insight into their own workload and make it easier to plan free time.
- **Push notifications for debts:** sending automatic reminders to students with debts, instead of tracking them manually through the Dashboard and Payments screens.
- **Data export:** the ability to export data (students, lessons, payments) to PDF or CSV format, for archiving or bookkeeping purposes.
- **Support for multiple teachers (multi-teacher):** extending the data model so that several teachers can independently use the same installation of the application, with appropriate data isolation per teacher.
- **Working without an internet connection (offline sync):** caching data locally on the device, so that the basic functionalities (reviewing students and lessons already loaded) remain available even without an active internet connection.

Taking all of the above into account, it can be concluded that the solution developed fully meets the functional and technical requirements set for the purposes of this bachelor's thesis, while at the same time leaving clearly defined and realistically achievable room for further improvement towards a production-ready, more widely applicable application. The value of the solution described in this thesis lies not only in the academic requirements fulfilled, but also in its proven applicability in a real context: the principle that the BaaS approach enables independent developers to build functional solutions without their own server infrastructure (Moroney, 2017), mentioned in the introductory part of the thesis as a theoretical basis, is confirmed here by an application that is in everyday use at the time of writing. Similarly, the attention paid to ease of use on a small screen (Nielsen & Budiu, 2012) was reflected in the positive reactions of the other teachers to whom the application was demonstrated, which rounds the thesis off as an example in which the academic and practical results directly confirm each other.

References

- [1] Banks, A., & Porcello, E. (2020) Learning React: Modern Patterns for Developing React Apps, 2nd edn, O'Reilly Media, Sebastopol
- [2] Boduch, A., & Derks, R. (2024) React and React Native: Build Cross-Platform Web and Mobile Applications Using React 18 and TypeScript, 5th edn, Packt Publishing, Birmingham
- [3] Eisenman, B. (2017) Learning React Native: Building Native Mobile Apps with JavaScript, 2nd edn, O'Reilly Media, Sebastopol
- [4] Expo (2026) Documentation [online]. Expo (650 Industries, Inc.). Retrieved from <https://docs.expo.dev> [Accessed 07.08.2026]
- [5] ExchangeRate-API (2026) API Documentation — Standard Requests [online]. Retrieved from <https://www.exchangerate-api.com/docs/standard-requests> [Accessed 07.08.2026]
- [6] Google LLC (2026) Firebase Documentation [online]. Retrieved from <https://firebase.google.com/docs> [Accessed 07.08.2026]
- [7] Moroney, L. (2017) The Definitive Guide to Firebase: Build Android Apps on Google's Mobile Platform, Apress, New York
- [8] Nielsen, J., & Budiu, R. (2012) Mobile Usability, New Riders, Berkeley
- [9] React Native (2026) Documentation [online]. Meta Platforms, Inc. Retrieved from <https://reactnative.dev/docs/getting-started> [Accessed 07.08.2026]
- [10] React Navigation (2026) Getting Started [online]. Retrieved from <https://reactnavigation.org/docs/getting-started/> [Accessed 07.08.2026]
- [11] Živković, M. (2024) Razvoj mobilnih aplikacija: Android Java programiranje [Mobile Application Development: Android Java Programming], 2nd edn, Singidunum University, Belgrade