UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Technical Report

SECJ3623: Mobile Application Programming

Semester I, Session 2025/2026

Project Title: EasyInvo

Lecturer: AP Dr. Mohd Shahizan Othman

Bil	Name	Matric No.
1.	ELAWARASI A/P NADARAJAN	SX211698ECRHF04
2.	JANAVI RADHA A/P BALA	SX211701ECRHF04
3.	AIMIE NATASYA BINTI AYUB	SX221610ECJHF04
4.	MUHAMMAD FAIZ FITRI BIN MOHD NOH	SX220354ECJHS04

Table of Contents

1. Development Approach and Workflow 4
   1.1. Development Methodology 4
   1.2. Development Workflow 4
   1.3. Project Structure 6
2. Software, Tools and Frameworks 9
   2.1. Core Technologies 9
   2.2. Third-Party Libraries & Their Purpose 10
   2.3. Supported Platforms 10
3. Database and Data Handling Design 11
   3.1. Understanding the Database Architecture 11
   3.2. How Data is Organized 11
   3.3. Data Models Explained 12
      3.3.1. Invoice Model - The Core Financial Document 12
      3.3.2. Client Model - Customer Information 14
      3.3.3. Item Model - Products and Services 14
      3.3.4. Transaction Model - Income Tracking 15
      3.3.5. Business Profile Model 15
   3.4. How Data is Stored and Retrieved 16
4. Implementation of CRUD Operations 17
5. Important Source Code Snippets 18
   5.1. Application Entry Point (main.dart) 18
   5.2. Authentication – User Login (Firebase Auth) 19
   5.3. Authentication Gate – Login State Control 19
   5.4. Invoice Calculation Logic (Core Business Logic) 20
   5.5. Invoice Item Model (Data Model) 20
   5.6. Object Serialization – Saving Data to Firestore 21
   5.7. Object Deserialization – Loading Data from Firestore 21

5.8. Create Invoice – Firestore Create Operation 22
5.9. Read Invoices – Real-Time Data Retrieval 23
5.10. Update Invoice Status (Paid / Unpaid) 23
6. Technical Challenges and Solutions 24
6.1. Challenge 1: Managing User Authentication State Across Multiple Platforms 24
6.2. Challenge 2: Ensuring Accurate and Consistent Invoice Calculations 24
6.3. Challenge 3: Handling Null and Missing Data from Firestore 25
6.4. Challenge 4: Real-Time Data Synchronization and UI Updates 25
6.5. Challenge 5: Secure User Data Isolation 26
6.6. Challenge 6: File Upload and Storage Management 26
6.7. Challenge 7: Cross-Platform UI Consistency 27

1. Development Approach and Workflow

1.1. Development Methodology

This project was developed using an Agile-based approach, which means the development process was divided into small, manageable phases where features were built incrementally and tested regularly. This methodology allows for flexibility and quick adjustments based on requirements.

The EasyInvo project was built following these fundamental principles:

- Modular Architecture: The code is organized into three main layers:
    - Service Layer: Contains all business logic and Firebase operations
    - Model Layer: Defines data structures for invoices, clients, items, and transactions
    - UI Layer: Presents information to users through screens and widgets
- Cloud-First Development: Instead of storing data locally on the phone, we use Firebase Cloud to store all data securely. This allows users to access their information from any device.
- User-Centric Iteration: The application was designed with real users in mind (freelancers, small business owners, service providers), and each feature was developed based on their actual needs.
- Cross-Platform Support: The application runs on 6 different platforms (iOS, Android, Web, Windows, macOS, Linux) without needing to rewrite the code for each platform.

1.2. Development Workflow

The development of EasyInvo was organized into five distinct phases, similar to how a construction project is built stage by stage:

Phase 1: Project Initialization & Setup During this phase, we set up the Flutter project and connected it to Firebase. Think of this as laying the foundation of a building. We:

- Created the Flutter project structure
- Configured Firebase (Google's cloud platform) to handle user authentication and data storage

- Designed the visual theme and color scheme for the entire application
- Set up the authentication system so users can sign up and log in securely

Phase 2: Core Infrastructure Development This phase focused on building the "backbone" of the application:

- Implemented the Firebase Authentication service (handles user sign-up and login)
- Created the Firestore database structure (organized like filing cabinets in the cloud)
- Designed data models (the structure of invoices, clients, items, and transactions)
- Built the service layer that communicates between the user interface and the cloud database

Phase 3: Feature Development This is where the main functionality was implemented:

- Invoice management system (create, edit, delete, mark as paid)
- Client relationship management (add, edit, delete, search)
- Item/inventory management (track products and services)
- Income tracking
- Business profile setup with logo upload
- Report generation in PDF and Excel formats

Phase 4: User Interface Implementation We created a visually appealing and easy-to-use interface:

- Implemented Material Design 3 (a modern design system by Google)
- Created 22 different screens for various tasks
- Designed responsive layouts that work on phone screens and large monitors
- Set up navigation between screens
- Created reusable UI components (buttons, input fields, cards)

Phase 5: Testing & Optimization Finally, we thoroughly tested the application:

- Tested on multiple devices and platforms
- Fixed bugs and errors
- Optimized performance (made the app run faster)

- Validated all calculations and data handling
- Cross-platform testing to ensure consistency

1.3. **Project Structure**

The project is organized in a folder structure that separates different concerns:

```
easyinvo/
├── lib/                          # Main application code
│   ├── main.dart                 # App entry point & configuration
│   ├── app_theme.dart            # Global theme definitions
│   ├── firebase_options.dart     # Firebase configuration
│   │
│   ├── models/                   # Data models
│   │   ├── transaction_model.dart
│   │   └── [other models]
│   │
│   ├── services/                 # Business logic & APIs
│   │   ├── auth_service.dart     # Authentication (Firebase Auth)
│   │   ├── business_service.dart # Business profile management
│   │   ├── client_service.dart   # Client/Customer data
│   │   ├── invoice_service.dart  # Invoice generation & management
│   │   ├── item_service.dart     # Product/Item management
│   │   ├── transaction_service.dart # Income/Expense tracking
│   │   └── [other services]
│   │
│   ├── widgets/                  # Reusable UI components
│   │   ├── auth_shell.dart       # Auth-related widgets
│   │   └── [other widgets]
│   │
│   └── screens/                  # UI Screens (organized by feature)
│       │
│       ├── auth/                 # Authentication screens
│       │   ├── auth_gate.dart    # Route controller (logged in/out)
```

```
│   │   ├── sign_in_screen.dart      # Login page
│   │   └── sign_up_screen.dart      # Registration page
│   │
│   ├── home/                        # Home/Dashboard
│   │   └── home_screen.dart                # Main dashboard & navigation hub
│   │
│   ├── business/                            # Business setup
│   │   └── business_setup_screen.dart       # Business profile configuration
│   │
│   ├── clients/                              # Customer management
│   │   ├── clients_screen.dart              # List all customers
│   │   ├── add_client_screen.dart           # Create new customer
│   │   ├── client_details_screen.dart       # View customer details
│   │   └── edit_client_screen.dart          # Edit customer info
│   │
│   ├── invoices/                            # Invoice management
│   │   ├── invoices_screen.dart             # List all invoices
│   │   ├── invoice_details_screen.dart      # View invoice details
│   │   └── invoice_management_screen.dart  # Create/Edit invoices
│   │
│   ├── items/                               # Product/Item management
│   │   ├── items_screen.dart                # List all items
│   │   ├── add_item_screen.dart             # Create new item
│   │   ├── item_details_screen.dart         # View item details
│   │   └── edit_item_screen.dart            # Edit item info
│   │
│   ├── transactions/                        # Income tracking
│   │   ├── income_screen.dart               # View income history
│   │   └── add_income_screen.dart           # Record new income
│   │
│   ├── reports/                             # Analytics & Reports
│   │   └── reports_screen.dart              # Financial reports & analysis
│   │
│   └── settings/                            # Settings & Help
```

```
│       ├── settings_screen.dart          # App settings
│       ├── help_support_screen.dart      # Help & support chatbot
│       └── about_screen.dart             # About app info
│
├── android/                              # Android native code
├── ios/                                  # iOS native code
├── web/                                  # Web platform
├── windows/                              # Windows platform
├── macos/                                # macOS platform
├── linux/                                # Linux platform
│
├── assets/                               # Static resources
│   ├── Icon/                             # App icons
│   └── templates/                        # Invoice templates
│
├── pubspec.yaml                          # Flutter dependencies
├── analysis_options.yaml                 # Linting rules
├── firebase.json                         # Firebase configuration
```

Why This Structure?

- Separation of Concerns: Each folder has a specific responsibility, making the code organized and easy to maintain
- Scalability: As the project grows, it's easy to add new features without affecting existing code
- Reusability: Services and widgets can be reused across different screens

2. Software, Tools and Frameworks

2.1. Core Technologies

This section explains the main tools and technologies used to build EasyInvo. Think of these as the essential materials and tools a builder uses to construct a house.

Table 1: Software, Tools and Frameworks

Technology	Version	What It Does
Flutter	Latest	A framework that lets us write code once and run it on Android, iOS, Web, Windows, Mac, and Linux
Dart	^3.9.2	The programming language we use to write the application code
Firebase Core	^4.2.0	Initializes and sets up the Firebase platform
Firebase Auth	^6.1.1	Manages user authentication (sign-up, login, password reset)
Cloud Firestore	^6.0.3	A real-time database stored in the cloud that syncs automatically
Firebase Storage	^13.0.4	Cloud storage for files like company logos and invoices

What is Flutter?

- Flutter is a mobile development framework created by Google. Instead of writing separate code for iPhone and Android, when write code once in Dart and it works on both. It's like writing a recipe that works in any kitchen, regardless of the equipment available.

What is Firebase?

- Firebase is Google's backend-as-a-service platform. It provides:
- Authentication: Secure user login system
- Database: Cloud storage for all the application data
- Storage: Cloud space for files
- Real-time Sync: Data updates automatically across all devices

2.2. **Third-Party Libraries & Their Purpose**

Beyond the core technologies, we used additional libraries (pre-written code) that extend the functionality:

For User Interface Design

- Material Design 3: A design system that makes the app look modern and professional
- Cupertino Icons (^1.0.8): Apple-style icons for iOS users

For Document Generation & Sharing

- PDF (^3.11.3): Creates professional-looking PDF invoices
- Printing (^5.14.2): Allows users to print or export documents as PDF
- Excel (^4.0.6): Generates Excel files for detailed reports
- Share Plus (^12.0.1): Lets users share documents via email, messaging, etc.
- Image Picker (^1.2.0): Allows users to select and upload company logos

For Utilities

- Intl (^0.20.2): Handles date formatting and multiple languages

For Development

- Flutter Lints (^5.0.0): Checks code quality and finds potential bugs
- Flutter Launcher Icons (^0.13.1): Helps create app icons for different platforms

2.3. **Supported Platforms**

One of the powerful features of Flutter is that the same code runs on multiple platforms:

Table 2:Support platforms

Platform	**Type**	**Purpose**
Android	Mobile	For smartphones and tablets running Android
iOS	Mobile	For iPhones and iPads
Web	Browser	For accessing via Chrome, Firefox, Safari
Windows	Desktop	For Windows computers
macOS	Desktop	For Apple computers
Linux	Desktop	For Linux operating systems

This means a freelancer can use EasyInvo on their phone while on the job, switch to a tablet while commuting, or use the desktop version in their office - all with the same data synchronized in real-time.

3. Database and Data Handling Design

3.1. Understanding the Database Architecture

The database is the "brain" of the application - it stores all the important data. EasyInvo uses Cloud Firestore, which is like a smart filing system in the cloud.

Why Cloud Firestore?

- Real-time Synchronization: When one user updates an invoice, all other users see the change instantly
- Scalability: Can handle thousands of users without slowing down
- Security: Built-in security rules ensure users only see their own data
- Offline Support: The app works even when there's no internet - it syncs when online again
- No Server Management: We don't have to worry about maintaining servers

3.2. How Data is Organized

Think of Cloud Firestore like a library. Instead of one big filing cabinet, it has separate collections (like different sections) where similar data is grouped together:

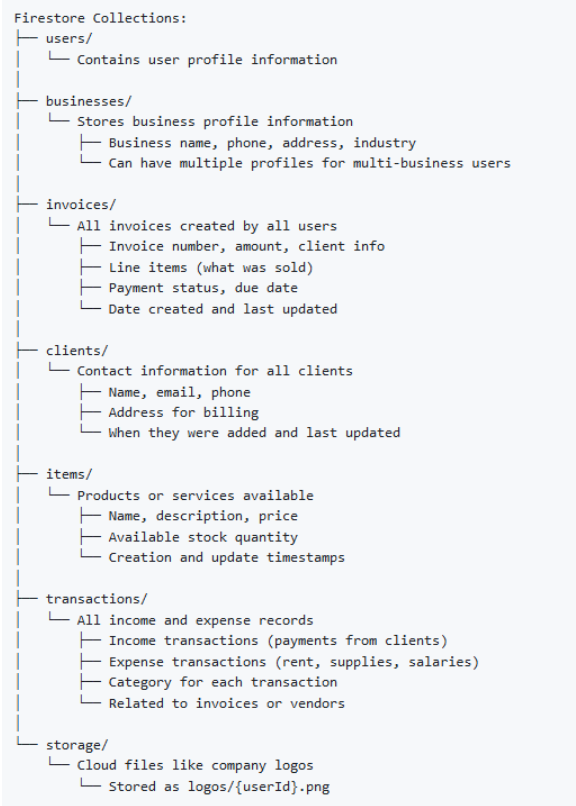

Figure 1: Data model

3.3. **Data Models Explained**

A data model is a blueprint that defines what information we collect and how it's structured. It's like a form that specifies which fields need to be filled out.

3.3.1. Invoice Model - The Core Financial Document

An Invoice is a bill sent to clients. Below is the complete data structure for storing all invoice information:

```dart
class Invoice {
  final String? id;                    // Unique document ID from database
  final String uid;                    // User ID who created this invoice
  final String referenceNumber;        // Unique invoice number (e.g., INV-2024-001)
  final String clientId;               // Client's unique identifier
  final String clientName;             // Client's name (denormalized for efficiency)
  final String? clientPhone;           // Client's phone number
  final String? clientAddress;         // Client's billing address
  final List<InvoiceItem> items;       // List of products/services sold
  final double subtotal;               // Sum of all line items (auto-calculated)
  final double discount;               // Discount amount applied
  final double tax;                    // Tax or GST amount
  final double total;                  // Final total (auto-calculated: subtotal - discount + tax)
  final DateTime dueDate;              // When payment is due
  final bool isPaid;                   // Payment status (true = paid, false = unpaid)
  final DateTime? paidDate;            // Date when payment was received
  final String notes;                  // Special notes or memo on the invoice
  final String paymentMethod;          // How the invoice was paid (cash, card, bank transfer, etc.)
  final double paidAmount;             // Amount actually received (for partial payments)
  final DateTime createdAt;            // When the invoice was created (system timestamp)
  final DateTime updatedAt;            // When the invoice was last modified (auto-updated)
  final String? businessId;            // Which business profile this invoice belongs to
  final String? businessName;          // Business name displayed on invoice
  final String? businessEmail;         // Business email for client contact
  final String? businessPhone;         // Business phone number
  final String? businessAddress;       // Business address for invoice header
}
```

Figure 2: Invoice Model

Why We Store All This Information:

Table 3: Invoice model info

Information	**Purpose**
id, uid	Identify the invoice and its owner for database storage
referenceNumber	Unique numbering system for invoice tracking and accounting
Client Data	Store client details with the invoice so it remains complete even if client profile is later modified or deleted
items, subtotal, total	Automatic calculations prevent math errors; totals computed once and stored for consistency
discount, tax	Track adjustments made to the invoice price
isPaid, paidDate, paidAmount	Complete payment tracking for accounting and follow-ups on unpaid invoices
createdAt, updatedAt	Audit trail showing when changes were made (important for compliance)
businessId/Details	Professional invoices display business information; supports multi-business users
notes, paymentMethod	Additional context for accounting and client communication

3.3.2.Client Model - Customer Information

A Client is a customer who receives invoices. We store essential contact information:

```
class Client {
  final String? id;          // Document ID
  final String uid;          // Owner UID
  final String name;
  final String email;
  final String phone;
  final String address;
  final DateTime createdAt;  // System timestamp
  final DateTime updatedAt;  // Auto-updated on changes
}
```

Figure 3:Client Model

Data Handling:

- Clients are automatically sorted alphabetically by name in the app
- Each user only sees their own clients (privacy protection)
- Timestamps are automatically managed by the system

3.3.3. Item Model - Products and Services

An Item represents a product or service that can be invoiced to clients:

```
class Item {
  final String? id;              // Document ID
  final String uid;              // Owner UID
  final String name;
  final String description;
  final double unitPrice;
  final int availableStock;      // Inventory tracking
  final DateTime createdAt;      // System timestamp
  final DateTime updatedAt;      // Auto-updated
}
```

Figure 4:Item Model

Inventory Management:

- Stock tracking prevents overselling
- Items can be reused across multiple invoices
- Prices are stored so they don't change when edited later

3.3.4. Transaction Model - Income Tracking

A Transaction records money coming in:

```
enum TransactionType { income, expense }

// Income comes from: client payments, bank interest, refunds, investments
enum IncomeCategory { invoicePayment, bankInterest, refund, investment, other }

class Transaction {
  final String? id;                // Document ID
  final String uid;                // Owner UID
  final TransactionType type;      // Income or expense
  final double amount;
  final DateTime date;             // Transaction date
  final String category;           // Categorized for reporting
  final String description;
  final String? paymentMethod;     // Cash, card, bank transfer, etc.
  final String? relatedTo;         // Reference to invoice or vendor
  final String? relatedToName;     // Related entity name for quick lookup
  final DateTime createdAt;        // System timestamp
  final DateTime updatedAt;        // Auto-updated
}
```

Figure 5:Transaction Model

Financial Analysis:

- The app can show total income
- Categorization helps identify from where money is coming

3.3.5. Business Profile Model

A Business Profile contains information about the user's business:

```
class BusinessProfile {
  final String? id;               // Unique identifier
  final String uid;               // Owner
  final String name;              // Business name
  final String address;           // Business address
  final String phone;             // Business phone
  final String industry;          // Industry type (helps categorization)
  final String? logoUrl;          // Link to company logo in cloud storage
}
```

Figure 6:Business Profile Model

Multi-Business Support:

- Users can create multiple business profiles
- Each profile has its own invoices and clients
- Logo is stored in Firebase Storage (separate from the database)

3.4. **How Data is Stored and Retrieved**

When Saving Data:

```
// Data is converted to a map (key-value pairs)
'invoiceNumber': 'INV-2024-001',
'clientName': 'John Smith',
'amount': 1500.00,
'date': Timestamp.fromDate(DateTime.now()),  // Stores exact time
'updatedAt': FieldValue.serverTimestamp()   // Server's time (not user's phone time)
```

Figure 7:Stored and Retrieved

Why Server Timestamps?

- Prevents cheating (user can't change the time)
- Ensures consistency across all devices
- Accurate for reporting and compliance

User Isolation for Security:

```
// Every query includes this filter
where('uid', isEqualTo: currentUser.uid)
```

Figure 8:User Isolation

This ensures users only see their own data, not other users' invoices or clients.

Real-time Updates: When a user creates an invoice, all their devices see the update instantly through a live data stream. It's like having a conversation where both people always see the latest messages.

Data Denormalization: We intentionally store client information inside each invoice (name, phone, address). This way:

- If the client moves and updates their address, old invoices still show the original address
- We don't need to look up client information separately
- The invoice is complete even if the client is later deleted

4. Implementation of CRUD Operations

CRUD stands for Create, Read, Update, and Delete - the four basic operations can do with any data. This section will explain how EasyInvo implements these operations for each type of data.

Table 4:CRUD Operations

Area	Create	Read	Update	Delete
Clients	Add clients with names, contacts, and addresses; each client is tied to the owning account.	Client lists update instantly and any client can be opened for details.	Change only the needed fields (for example, a new phone number) while keeping other details intact.	Removing a client affects only that record; past invoices keep their embedded client details for history.
Items	Add items with name, description, price, and available stock.	Item lists refresh live and any item can be opened for details.	Adjust price, description, or stock without affecting other item data.	Remove items from the catalogue. Past invoices still show the item details used at the time.
Invoices	Create invoices by selecting a client and adding line items. Subtotal, discount, tax, and total are calculated automatically.	Live, sorted lists allow quick filtering of unpaid items and any invoice can be opened for full details.	Edit invoices as needed and marking an invoice as paid records the received amount and date.	Remove erroneous invoices/embedded client and business info keep records clear and consistent.
Transactions	Record income (payments) and expenses with categories and payment methods, linked to invoices when relevant.	View all transactions in real time or focus on income by payment method	Adjust amounts and details while keeping the timeline accurate.	Remove incorrect entries, totals and summaries update based on remaining data.

Business Profiles	Create profiles by uploading a logo and entering name, address, phone, and industry. Multiple profiles per account are supported.	The app stays in sync with the active business profile and the list of profiles.	Select a profile, modify details, and save changes so they appear across invoices and reports.	Choose a profile to remove and the system confirms and then deletes it without affecting the account. Logos are stored securely online and reused across documents.

5. Important Source Code Snippets

This section presents the most important source code snippets used in the EasyInvo application. These snippets represent critical system functionalities such as authentication, invoice calculation, CRUD operations, database interaction, file storage, and real-time updates. Each snippet is accompanied by an explanation to demonstrate understanding of how the system was implemented.

5.1. Application Entry Point (main.dart)

This file initializes Firebase and launches the application.

Code Snippet:

```dart
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform);
  runApp(const MyApp());
}
```

Figure 9:Application Entry Point

Explanation:

- WidgetsFlutterBinding.ensureInitialized() prepares Flutter before Firebase initialization.
- Firebase.initializeApp() connects the app to Firebase services.
- This ensures authentication and database services are ready before the UI loads.

5.2. Authentication – User Login (Firebase Auth)

Handles secure user login using Firebase Authentication.

Code Snippet:

```dart
Future<User?> signIn(String email, String password) async {
  final cred = await _auth.signInWithEmailAndPassword(
    email: email.trim(),
    password: password,
  );
  return cred.user;
}
```

Figure 10:User Login

Explanation:

- Validates user credentials against Firebase Authentication.
- Ensures only registered users can access the system.
- Firebase manages password security and encryption automatically.

5.3. Authentication Gate – Login State Control

Controls whether the user sees the login screen or main dashboard.

Code Snippet:

```dart
@override
Widget build(BuildContext context) {
  return StreamBuilder<User?>(
    stream: AuthService().authState,
    builder: (context, snap) {
      if (snap.connectionState == ConnectionState.waiting) {
        return const Scaffold(
          body: Center(child: CircularProgressIndicator()),
        ); // Scaffold
      }
      if (snap.data != null) {
        return const HomeScreen();
      }
      return const SignInScreen();
    },
  ); // StreamBuilder
}
}
```

Figure 11:Login State Control

Explanation:

- Listens to authentication state changes.
- Automatically redirects users based on login status.
- Improves security and user experience.

5.4. Invoice Calculation Logic (Core Business Logic)

Ensures accurate financial calculations for invoices.

Code Snippet:

```
InvoiceItem({
  required this.itemId,
  required this.name,
  required this.description,
  required this.unitPrice,
  required this.quantity,
}) : total = unitPrice * quantity;
```

Figure 12:Invoice Total amount Calculation Logic

```
subtotal = items.fold(0.0, (sum, item) => sum + item.total),
total =
    items.fold(0.0, (sum, item) => sum + item.total) - discount + tax,
createdAt = createdAt ?? DateTime.now(),
updatedAt = updatedAt ?? DateTime.now();
```

Figure 13: Tax and Discount calculation logic

Explanation:

- fold() calculates the sum of all invoice items.
- Prevents manual calculation errors.
- Calculations are performed automatically whenever an invoice is created or updated.
- This calculation includes deduction amount of discount

5.5. Invoice Item Model (Data Model)

Defines the structure of each invoice line item.

Code Snippet:

```
class InvoiceItem {
  final String itemId;
  final String name;
  final String description;
  final double unitPrice;
  final int quantity;
  final double total;

  InvoiceItem({
    required this.itemId,
    required this.name,
    required this.description,
    required this.unitPrice,
    required this.quantity,
  }) : total = unitPrice * quantity;
```

Figure 14:Invoice Item Model

Explanation:

- Calculated once when object is created
- Stored as a field
- Faster for repeated access
- Immutable (can't change)

5.6. Object Serialization – Saving Data to Firestore

Converts Dart objects into Firestore-compatible format.

Code Snippet:

```dart
Map<String, dynamic> toMap() => {
  'itemId': itemId,
  'name': name,
  'description': description,
  'unitPrice': unitPrice,
  'quantity': quantity,
  'total': total,
};
```

Figure 15:Saving Data to Firestore

Explanation:

- Converts objects into key–value pairs.
- Required for Firestore storage.
- Promotes consistent data structure.

5.7. Object Deserialization – Loading Data from Firestore

Safely reconstructs objects from database data.

Code Snippet:

```dart
static InvoiceItem fromMap(Map<String, dynamic> m) {
  return InvoiceItem(
    itemId: m['itemId'] ?? '',
    name: m['name'] ?? '',
    description: m['description'] ?? '',
    unitPrice: (m['unitPrice'] ?? 0).toDouble(),
    quantity: (m['quantity'] ?? 0).toInt(),
  );
}
```

Figure 16:Loading Data from Firestore

Explanation:

- Uses null safety to prevent crashes.
- Handles missing or invalid database fields.
- Ensures backward compatibility.

5.8. Create Invoice – Firestore Create Operation

Stores new invoice data in the database.

Code Snippet:

```dart
class InvoiceService {
  final _auth = FirebaseAuth.instance;
  final _col = FirebaseFirestore.instance.collection('invoices');
```

Figure 17:Firestore Create Operation

```dart
// Add a new invoice
Future<String> addInvoice(Invoice invoice) async {
  final docRef = await _col.add(invoice.toMap());
  return docRef.id;
}

// Update an existing invoice
Future<void> updateInvoice(String invoiceId, Invoice invoice) async {
  await _col.doc(invoiceId).update(invoice.toMap());
}

// Delete an invoice
Future<void> deleteInvoice(String invoiceId) async {
  await _col.doc(invoiceId).delete();
}

// Mark invoice as paid
Future<void> markAsPaid(String invoiceId) async {
  await _col.doc(invoiceId).update({
    'isPaid': true,
    'paidDate': FieldValue.serverTimestamp(),
    'updatedAt': FieldValue.serverTimestamp(),
  });
}

// Mark invoice as unpaid
Future<void> markAsUnpaid(String invoiceId) async {
  await _col.doc(invoiceId).update({
    'isPaid': false,
    'paidDate': null,
    'updatedAt': FieldValue.serverTimestamp(),
  });
}
```

Figure 18: Invoice CRUD

Explanation:

- Adds a new, update and delete invoice document.
- Associates data with the logged-in user.
- Uses server time for accuracy and audit purposes.

5.9. Read Invoices – Real-Time Data Retrieval

Fetches invoices using Firestore real-time streams.

Code Snippet:

```dart
Stream<List<Invoice>> watchAllInvoices() {
  return _col.snapshots().map((snap) {
    final invoices = snap.docs.map((doc) => Invoice.fromDoc(doc)).toList();
    invoices.sort((a, b) => b.createdAt.compareTo(a.createdAt));
    return invoices;
  });
}
```

Figure 19:Real-Time Data Retrieval

Explanation:

- Retrieves only the user's invoices.
- Updates UI instantly when data changes.
- Enhances responsiveness and usability.

5.10. Update Invoice Status (Paid / Unpaid)

Marks invoices as paid and records payment details.

Code Snippet:

```dart
// Mark invoice as paid
Future<void> markAsPaid(String invoiceId) async {
  await _col.doc(invoiceId).update({
    'isPaid': true,
    'paidDate': FieldValue.serverTimestamp(),
    'updatedAt': FieldValue.serverTimestamp(),
  });
}

// Mark invoice as unpaid
Futu          sUnpaid(String invoiceId) async {
  aw  Type: String  voiceId).update({
    'isPaid': false,
    'paidDate': null,
    'updatedAt': FieldValue.serverTimestamp(),
  });
}
```

Figure 20:5.Update Invoice Status

Explanation:

- Updates existing invoice data.
- Records payment time securely.
- Supports financial tracking and reporting.

6. Technical Challenges and Solutions

During the development of the EasyInvo mobile application, several technical challenges were encountered. These challenges are common in real-world mobile and cloud-based application development. This section discusses the key issues faced and the solutions implemented to overcome them.

6.1. Challenge 1: Managing User Authentication State Across Multiple Platforms

Problem:

EasyInvo supports multiple platforms (mobile, web, and desktop). Managing user authentication consistently across different platforms was challenging, especially when handling login, logout, and session persistence.

Solution:

Firebase Authentication was integrated as the centralized authentication service. An authentication gate was implemented using real-time authentication state listeners. This allowed the application to automatically detect login state changes and route users accordingly.

Outcome:

- Seamless login experience across platforms
- Improved security and session management
- Reduced authentication-related bugs

6.2. Challenge 2: Ensuring Accurate and Consistent Invoice Calculations

Problem:

Invoice totals must be accurate at all times. Manual calculations or scattered logic across multiple screens could lead to inconsistencies and financial errors.

Solution:

All invoice calculations (subtotal, discount, tax, and total) were centralized in the invoice model layer. This ensured that calculations were performed once and reused across the system.

Outcome:

- Eliminated duplicate calculation logic
- Prevented financial inconsistencies

- Improved code maintainability

6.3. Challenge 3: Handling Null and Missing Data from Firestore

Problem:

Data retrieved from Firestore may contain missing fields, null values, or unexpected data types, especially when the database schema evolves over time.

Solution:

Defensive programming techniques were applied using Dart's null-safety features. Default values and null-aware operators (??, ?) were used when deserializing data from Firestore.

Outcome:

- Prevented application crashes
- Improved stability and reliability
- Supported backward compatibility for older data

6.4. Challenge 4: Real-Time Data Synchronization and UI Updates

Problem:

Ensuring that the user interface updates immediately when invoices, clients, or items change was challenging, particularly when multiple devices were used simultaneously.

Solution:

Firestore real-time streams (snapshots()) were used to listen for database changes. UI components were built using reactive widgets that automatically update when data changes.

Outcome:

- Real-time updates across all devices
- Enhanced user experience
- No need for manual refresh actions

6.5. Challenge 5: Secure User Data Isolation

Problem:

In a multi-user system, it is critical to ensure that users can only access their own invoices, clients, and business data.

Solution:

All Firestore queries were filtered using the authenticated user's unique user ID (uid). This ensured strict data separation between users. Firebase security rules further enforced access control at the database level.

Outcome:

- Improved data privacy and security
- Prevented unauthorized data access
- Compliance with cloud security best practices

6.6. Challenge 6: File Upload and Storage Management

Problem:

Storing business logos and document files directly in the database is inefficient and can degrade performance.

Solution:

Firebase Storage was used to store files such as business logos. Only the file URL was saved in Firestore, keeping the database lightweight and efficient.

Outcome:

- Improved performance and scalability
- Efficient separation of data and files
- Simplified file management

6.7. **Challenge 7: Cross-Platform UI Consistency**

Problem:

Ensuring that the user interface looked consistent and usable across different screen sizes and platforms was challenging.

Solution:

Flutter's responsive layout widgets and Material Design 3 principles were applied. Reusable UI components were created to maintain consistent design across all screens.

Outcome:

- Consistent user experience across platforms
- Reduced UI-related bugs
- Easier maintenance and future enhancements