Latvijas Universitāte

Faculty of Computer Science

Course code and title: DatZK006: Kvalifikācijas projekts

LANGUAGE SCHOOL BILLING SYSTEM

QUALIFICATION PAPER

Author: **Iļja Junkins**

Student's ID card No: **ij23031**

Advisor: **Zane Bičevska**

Rīga 2026

Table of Contents

1. Introduction

1.1 Relevance and Problem Statement

Language schools, especially small and medium-sized educational institutions, face serious operational challenges when performing administrative tasks related to billing students and maintaining financial records.

For example, a survey conducted by SAP Concur and Kelton Global [1] indicates that large educational organizations may spend thousands of working hours per month on manual invoice and expense processing. While small and medium-sized language schools operate on a much smaller scale, similar inefficiencies arise due to limited administrative resources and reliance on manual or semi-manual workflows.

In many language schools, billing processes are still handled using spreadsheets or ad-hoc tools without centralized data control. Such approaches increase the risk of data inconsistencies, delayed invoicing, calculation errors, and limited traceability of payments. As the number of students, courses, and billing models grows, these manual solutions become increasingly difficult to maintain and scale.

Existing billing solutions has several significant limitations

- **Limitations of off-the-shelf software:** Existing accounting software is designed for general business use and lacks the features needed by language schools, such as per-lesson billing, attendance-based billing, and flexible subscription models.

- **Cost barriers:** Professional invoicing systems typically require expensive subscriptions (€50–200 per month) and cloud infrastructure, making them economically unviable for small language schools operating on a limited budget.
- **Data privacy concerns:** Cloud solutions raise concerns about data protection, especially in the European Union, where GDPR compliance is mandatory. Language schools that handle students' personal information need local data storage options.

Complexity versus ease of use: Available solutions either oversimplify (e.g., spreadsheets) or overcomplicate (enterprise ERP systems) the billing process, failing to address the specific workflow requirements of language school administrators.

The language school billing system solves these problems by providing a specially designed single-user desktop application specifically tailored to the billing workflows of language schools, eliminating the need for expensive

1.2 Goal

The goal of this qualification project is to design, develop, and validate a desktop billing management system specifically designed for small and medium-sized language schools, which allows for effective tracking of student enrollment, issuing invoices based on attendance, automatically generate sequentially numbered invoices, and manage payments—all in a secure environment with the ability to operate offline and store data locally.

1.3 Objectives

The following specific objectives were defined to achieve the stated goal:

1. **Requirements analysis:** Analyze the language school's billing workflows to determine the functional and non-functional requirements for the system.
2. **System architecture design:** Develop a multi-level software architecture that implements the separation of tasks between the presentation, business logic, and data access layers using modern design patterns.
3. **Implementation of basic functions:**
    - Management of students and courses with activation status tracking.
    - Enrollment management with support for both one-time and subscription billing modes.
    - Monthly attendance tracking with data integrity control (month blocking).
    - Automatic creation of draft invoices based on attendance records.
    - Sequential invoice numbering (format: PREFIX-YYYYMM-SEQ).
    - Creation of invoices in PDF format.

- Payment tracking with automatic balance calculation.

4. **Testing and validation:** Development and execution of comprehensive testing procedures, including unit tests, manual testing scenarios, and security checks.

1.4 Methods

The following methodologies and technical approaches were used in developing the billing system for the language school:

Development Methodology:

- **The system was developed in stages:** functions were added gradually, and after each stage, functional testing was conducted, based on which improvements were made.
- **Feature-Driven Development:** Development focused on implementing a full set of user-facing features in order of priority (student management → attendance tracking → invoice generation → payment tracking).

Software Engineering Practices:

- **Version Control:** A Git-based version control system hosted on GitHub (https://github.com/Uvlazhnitel/Language-School-Billing) with a feature branching workflow.
- **Code Generation:** Using the ent framework to automatically generate type-safe database access code from entity schemas, reducing boilerplate code and potential errors.
- **Design Patterns:** Using the service layer pattern, the data transfer object (DTO) pattern, the repository pattern (via ent ORM), and the singleton pattern for configuration management.

Testing Approach:

Unit Testing: Implemented table-based tests for validation functions to achieve 100% code coverage for critical input validation logic.

Manual Testing: Systematic manual testing of complete user workflows with documented test scenarios (over 30 test cases).

Technology Selection Rationale:

- **Go Language:** Chosen for the backend due to its high type safety, excellent performance, built-in concurrency support, and cross-platform compilation capabilities.
- **Wails Framework:** Selected for developing desktop applications using web technologies while maintaining native app performance and avoiding browser overhead.

- **ent ORM:** Selected for type-safe database operations and automatic migration generation from schema definitions.
- **SQLite:** Selected for data storage due to its zero-configuration requirements, single-file portability, and suitability for single-user desktop applications.
- **React + TypeScript:** Selected for UI development due to its component-based architecture and strong type checking.

1.5 Data and Fact Sources

The development and validation of this system are based on the following data sources and factual foundations:

Primary Sources:

- **Source Code Repository**: GitHub repository containing ~2,800 lines of production code (98 Go files, 12 TypeScript/TSX files)
  - Location: https://github.com/Uvlazhnitel/Language-School-Billing
- **Existing Test Suite**: Unit test suite with 19 test cases covering validation logic
  - Location: internal/validation/validate_test.go
  - Coverage: 100% of validation package
- **Database Schema**: ent entity schemas defining 9 core entities
  - Location: ent/schema/*.go (student.go, course.go, enrollment.go, invoice.go, invoiceline.go, payment.go, attendancemonth.go, settings.go, priceoverride.go)

Technical Documentation Sources:

- Framework Documentation: Wails v2 official documentation (https://wails.io/docs/)
- ent Framework: ent ORM documentation (https://entgo.io/docs/)
- Go Language Specification: Official Go documentation (https://go.dev/ref/spec)
- TypeScript Documentation: Official TypeScript handbook (https://www.typescriptlang.org/docs/)

Development Tools and Standards:

- **Go Version**: 1.24.0 (as specified in go.mod)
- **Wails Version**: v2.10.2
- **React Version**: 18.3
- **TypeScript Version**: 5.7

- **Build Tools**: Wails CLI, Go modules, npm package manager

Security Standards:

- **OWASP Top Ten**: Reference for web application security risks and mitigation strategies
- **HTML Escaping**: Standard HTML entity encoding for XSS prevention
- **Parameterized Queries**: SQL injection prevention via ent ORM's query builder

2. Software Requirements Specification

2.1 Purpose and Scope

Purpose: This section outlines the functional and non-functional requirements for the Language School Billing System, a desktop application designed to automate administrative billing processes in small and medium-sized language schools.

Scope:

Included functionality:

- Student information management (creation, updating, activation/deactivation, deletion with restrictions)
- Course management with customizable pricing (per lesson and subscription)
- Student enrollment in courses with flexible payment options (per lesson or subscription)
- Automatic generation of invoice drafts based on attendance records and subscription registration
- Invoicing with sequential numbering (format: PREFIX-YYYYMM-SEQ)
- Creation of PDF invoices.
- Payment accounting and automatic balance calculation
- Debtor identification and reporting

System limitations:

- The application runs completely offline on the local computer.
- Data is stored exclusively in a local SQLite database (~/LangSchool/Data/app.sqlite).
- PDF files are saved to the local file system (~/LangSchool/Invoices/YYYY/MM/).
- There are no external API integrations or network dependencies.

2.2 Users and Stakeholders

Primary User:

Language School Administrator: The person responsible for managing student billing, attendance tracking, and financial records. Typically, this is the school owner or a designated member of the administrative staff.

User Characteristics:

- Basic computer literacy (proficiency with desktop applications)
- Familiarity with language school operations and billing processes
- No programming or technical knowledge required

Stakeholder	Role	Interest	Influence
School Owner / Administrator	Primary user	Efficient billing, accurate records, time savings	High
Students	Indirect beneficiary	Accurate invoicing, clear billing information	Low
Accountant / Tax Authority	Data consumer	Accurate financial records, sequential invoice numbering	Medium

User Needs:

1. Reduce time spent on manual invoicing (goal: reduce from 15-20 hours per month to <5 hours per month)
2. Eliminate errors in manual calculations
3. Maintain organized financial documentation with sequential invoice numbering
4. Create professional PDF invoices with the school's branding
5. Track student payment status and identify outstanding accounts

2.3 Operating Environment and Constraints

Technical Environment:

Supported Operating Systems:

- Windows 10 or later (64-bit)
- macOS 11 (Big Sur) or later
- Linux distributions with modern kernel (Ubuntu 20.04+, Fedora 35+, or equivalent)

Minimum System Requirements:

- CPU: Dual-core processor, 1.5 GHz or faster
- RAM: 4 GB
- Disk Space: 100 MB for application + storage for database and PDFs
- Display: 1024x768 resolution minimum

Recommended System Requirements:

- CPU: Quad-core processor, 2.5 GHz or faster
- RAM: 8 GB
- Disk Space: 1 GB free space
- Display: 1920x1080 resolution or higher

Software Dependencies:

- No external software required (self-contained application)

Runtime Environment:

- Wails v2 runtime (embedded in application binary)
- Go 1.24.0 runtime (compiled into application)
- SQLite 3 (embedded database engine)
- No web browser required (native desktop application)

Constraints:

Technical Constraints:

- **TC-01**: Single-user only (no concurrent access support)
- **TC-02**: No network connectivity features (offline operation only)

Business Constraints:

- **BC-01**: Zero licensing costs (open-source components only)
- **BC-02**: No subscription fees or recurring costs
- **BC-03**: No external service dependencies to minimize operational costs

Regulatory Constraints:

- **RC-01**: Local data storage for GDPR compliance consideration (personal data remains on user's machine)
- **RC-02**: Sequential invoice numbering for accounting compliance

Design Constraints:

- **DC-01**: Must use Wails v2 framework (architectural decision)
- **DC-02**: Must use SQLite for data storage (portability requirement)
- **DC-03**: Must use Go language for backend (performance and type safety requirement)

- **DC-04**: Must use React + TypeScript for frontend (maintainability requirement)
- **DC-05**: Must generate PDF invoices (business requirement)

2.4 Assumptions and dependencies

Assumptions:

User Assumptions:

- **A-01:** The user has basic computer literacy and can operate desktop applications.
- **A-02:** User understands language school billing workflows (per lesson or subscription models)
- **A-03:** The user has administrative access to the local computer (to create a directory).
- **A-04:** The user can manually place the font files in the specified directory.
- **A-05:** User manually backs up data (copies SQLite file).

Technical assumptions:

- **A-06:** The operating system provides stable file system access for the database and PDF file storage.
- **A-07:** The system has enough disk space to expand the database and PDF archive.
- **A-08:** Antivirus software does not interfere with access to database files.
- **A-09:** System date/time is set correctly (for counting date and sequential numbering)

Operating assumptions:

- **A-10:** The school operates on a monthly billing cycle (billings are issued monthly).
- **A-11:** Account prefixes remain unchanged (change infrequently).
- **A-12:** Maximum expected data volume: 1000 students, 100 courses, 10,000 accounts per year.
- **A-13:** Single school location (multiple branch support not required)

Framework Dependencies:

- **D-02**: Wails v2.10.2 framework (embedded in application)
- **D-03**: Go 1.24.0 runtime (compiled into binary)
- **D-04**: ent v0.14.5 ORM framework

- **D-05**: React 18.3 UI library
- **D-06**: TypeScript 5.7 compiler

Library Dependencies:

- **D-07**: gofpdf library for PDF generation
- **D-08**: go-sqlite3 driver for database access
- **D-09**: Vite 6.0 for frontend build process

Platform Dependencies:

- **D-10**: Operating system file system APIs (directory creation, file I/O)
- **D-11**: Operating system window management (for desktop application)

Development Tool Dependencies (not required for end users):

- **D-12**: Wails CLI for building application
- **D-13**: Go compiler for backend compilation
- **D-14**: Node.js and npm for frontend build
- **D-15**: Git for version control
-

2.5 Functional Requirements

The following functional requirements are organized by subsystem. Each requirement includes a unique identifier (FR-X), description, and acceptance criteria.

FR-1: Student Information Management

Description: System will allow creating and managing student records with validation and sanitization.

Acceptance Criteria:

- User can create student with full name (required), phone (optional), email (optional), note (optional)
- System validates that full name is non-empty
- Student appears in student list after creation
- User can update student information and toggle active status
- System prevents deletion of students with existing enrollments or invoices

FR-2: Course Management

Description: The system must support the creation of courses with customizable pricing for various course types.

Acceptance Criteria:

- The user can create courses of the "group" or "individual" type.

- The user can set the lesson price and subscription price (non-negative floating-point values).
- The system verifies that prices are non-negative.
- The user can update and delete courses.
- The system prevents the deletion of courses with an active enrollment.
- Course prices are used for invoicing.

FR-3: Enrollment Management

Description: The system should allow students to enroll in courses with flexible payment settings.

Acceptance Criteria:

- The user can create enrollment records by linking a student to a course.
- The user can select the payment mode: "per_lesson" or "subscription."
- The user can set the discount percentage (0-100%).
- The system validates the discount percentage range.
- The system prevents duplicate enrollment records (the same student-course pair).
- Enrollment settings affect invoice generation.

FR-4: Monthly Attendance Tracking

Description: The system must track class attendance for each student-course pair with data integrity control.

Acceptance Criteria:

- The user can view an attendance grid organized by month.
- The user can edit the number of classes for each student-course-month combination.
- The user can use the "Add +1 to All" function to bulk update attendance data.
- Attendance data is used to calculate invoices for classes.

FR-5: Generating Draft Invoices

Description: The system must automatically generate draft invoices based on attendance and course registration data.

Acceptance Criteria:

- The user can initiate the creation of draft invoices for a specific year and month.

- The system creates draft invoices for all students with an active enrollment.
- For individual lessons: Total = Number of lessons x Lesson Price x (1 - Discount/100).
- For subscriptions: Total = Subscription Price x (1 - Discount/100).
- Drafts can be reviewed before sending.
- The system calculates the total amount based on individual invoice lines.

FR-6: Sequential Invoicing

Description: The system should issue sequential invoices in the format PREFIX-YYYYMM-SEQ.

Acceptance Criteria:

- The user can create individual invoice drafts
- The user can create batches of invoice drafts
- The system assigns a sequence number upon creation (e.g., LS-202512-001)
- The system maintains a sequential order without spaces
- The system increments the sequence counter after each creation
- Created invoices cannot be modified (immutable)
- The system changes the invoice status from "draft" to "created"

FR-7: Generating PDF Invoices

Description: The system must generate PDF invoices containing organization information

Acceptance Criteria:

- The PDF file is automatically generated when the invoice is issued.
- The PDF file contains the organization name and address (if configured).
- The PDF file contains the invoice number, date, and student information.
- The PDF file contains a table with invoice lines (description, quantity, unit price, amount). The PDF file displays the total amount.
- The PDF file is saved in ~/LangSchool/Invoices/YYYY/MM/NUMBER.pdf

FR-8: Recording Payments

Description: The system should allow recording payments with automatic invoice status updates.

Acceptance Criteria:

- The user can record payments by specifying the amount, payment method (cash/bank), and date.

- The user can link the payment to a specific invoice (optional).
- The user can add a note to the payment record.
- The system validates the payment amount as > 0.
- When a payment is linked to an invoice and the total payment amount ≥ the invoice amount, the system automatically changes the invoice status to "paid."
- The payment record is saved and displayed in the payment list.

FR-9: Balance Calculation and Accounts Receivable Tracking

Description: The system must calculate student balances and identify debtors.

Acceptance Criteria:

- The system calculates the balance = Σ(invoice amount, where the status is IN (issued, paid)) - Σ(payment amount).
- The user can view the balance for individual students.
- The user can view a list of debtors (students with a negative balance).
- The list of debtors displays the student name and balance amount.
- Balance calculations are mathematically correct.

FR-10: Parameter Configuration

Description: The system must allow customization of organizational data and invoice parameters.

Acceptance Criteria:

- The user can specify the organization name and address.
- The user can configure the invoice prefix (default: "LS").
- The system saves the next sequential number for invoice numbering.
- The settings are saved and remain in effect after restarting the application.
- The organization information is displayed in generated invoices.

2.6 Non-Functional Requirements

NFR-1: Performance - Application Startup

Description: The application should start quickly on standard hardware.

Requirement: Startup time ≤ 5 seconds from startup to user interface readiness.

Measurement: Time measured from the start of a process to the display of a window.

Priority: Medium.

NFR-2: Performance - User Interface Responsiveness

Description: User interface operations must be responsive.

Requirement: User interface operations must complete within 1 second.

Measurement: Time from user action (button click) to user interface update.

Priority: High.

Applicable operations: Loading student list, loading course list, submitting forms, updating attendance grid.

NFR-3: Performance - PDF Generation

Description: Generating PDF files should not cause noticeable delays.

Requirement: Generate PDF files ≤ 3 seconds per count

Measurement: Time from create command to PDF file save

Priority: Medium

NFR-4: Performance - Scalability

Description: The system must efficiently handle expected data volumes.

Requirement: Support at least 1000 students with query execution times < 1 second

Measurement: Database query execution time

Priority: Medium

NFR-5: Usability - Intuitive Interface.

Description: The application must be easy to use for target users.

Requirement: The user can perform basic tasks without documentation.

Measurement: Task completion by a new user without training.

Priority: High.

NFR-6: Usability - Error Messages.

Description: Error messages must be clear and provide information about possible actions.

Requirement: Error messages must explain what went wrong and how to fix it.

Examples: "Student name cannot be empty," "Cannot delete a student with an active registration."

Priority: High.

NFR-7: Reliability - Data Integrity.

Description: The system must maintain data consistency.

Requirement: Use database transactions for multi-stage operations.

Implementation: Support for ORM transactions.

Examples: Invoice issuance (invoice update + increment sequence) must be atomic.

Priority: Critical.

NFR-8: Reliability - Input Data Validation.

Description: The system must validate all user data before processing.

Requirement: 100% of user data must be validated before database operations.

Coverage: Test coverage of the validation suite = 100%.

Priority: High.

NFR-9: Robustness - Error Handling.

Description: The system must handle errors gracefully and without failure.

Requirement: All errors must be caught and logged, with user-friendly messages displayed.

Priority: High

NFR-10: Maintainability - Code Quality

Description: Code must comply with programming language best practices.

Requirement: Code must be linter-safe (golangci-lint for Go, ESLint for TypeScript).

Implementation: Configured linters in the project.

Priority: Medium

NFR-11: Maintainability - Code Organization

Description: Code must be organized with a clear separation of concerns. **Requirement**: Multi-tier architecture (presentation, application, business logic, data access).

Check: Clear package boundaries in the repository structure.

Priority: High

NFR-12: Maintainability - Documentation

Description: Code and system must be documented.

Requirement: Package-level documentation, function comments for public APIs

Coverage: Comprehensive documentation in Markdown format

Priority: High

NFR-16: Portability - Cross-Platform Support

Description: Application shall run on multiple operating systems. **Requirement**: Builds and runs on Windows 10+, macOS 11+, Linux (Ubuntu 20.04+)

Implementation: Wails v2 framework provides cross-platform support
Priority: High

NFR-17: Portability - No Platform-Specific Code

Description: Codebase shall avoid platform-specific dependencies. **Requirement**: Use cross-platform libraries and APIs only **Verification**: Single codebase compiles for all target platforms **Priority**: High

2.7 Use Cases and User Stories

Use Case 1: Complete Monthly Billing Cycle

Actor: Language School Administrator **Goal**: Generate and issue invoices for all students for completed month **Preconditions**: Students enrolled in courses, attendance recorded for month

Main Flow:

1. Administrator navigates to Attendance tab
2. Administrator reviews attendance records for month
3. Administrator navigates to Invoices tab
4. Administrator selects year and month
5. Administrator clicks "Generate Drafts"
6. System creates draft invoices for all students based on:
    - Attendance records (for per-lesson billing)
    - Active subscriptions (for subscription billing)
7. Administrator reviews draft invoices for accuracy
8. Administrator clicks "Issue All" to issue all draft invoices
9. System assigns sequential numbers to invoices
10. System generates PDF files for all invoices
11. System saves PDFs to organized directory structure
12. Administrator distributes invoice PDFs to students (manual process)

Postconditions: All invoices issued with sequential numbers, PDFs generated and saved

Use Case 2: Record Student Payment and Update Balance

Actor: Language School Administrator **Goal**: Record received payment and update student's account status **Preconditions**: Student has issued invoice(s)

Main Flow:

1. Administrator receives payment from student (cash or bank transfer)
2. Administrator navigates to Invoices tab
3. Administrator clicks "Open" on invoice
4. Administrator clicks "Record Payment"
5. Administrator enters payment amount
6. Administrator selects payment method (cash or bank)
7. Administrator enters payment note (optional)
8. Administrator clicks "Record Payment"
9. System validates payment amount > 0
10. System creates payment record
11. System checks if linked invoice is fully paid
12. If total payments ≥ invoice amount, system updates invoice status to "paid"
13. System displays updated payment list and student balance

Postconditions: Payment recorded, invoice status updated if fully paid, balance calculated

Use Case 3: Set Up New Course and Enroll Students

Actor: Language School Administrator

Goal: Add new course offering and enroll students

Preconditions: Students exist in system

Main Flow:

1. Administrator navigates to Courses tab
2. Administrator clicks "Add Course"
3. Administrator enters course name (e.g., "English A2 Group")
4. Administrator selects course type "group"
5. Administrator sets lesson price (e.g., 5.00 EUR)
6. Administrator sets subscription price (e.g., 40.00 EUR per month)
7. Administrator clicks "Save"
8. System validates prices ≥ 0
9. System creates course record
10. Administrator navigates to Enrollments tab
11. For each student to enroll:
    - Administrator clicks "Add Enrollment"
    - Administrator selects student
    - Administrator selects newly created course
    - Administrator selects billing mode (per-lesson or subscription)
    - Administrator sets discount percentage (0-100%, default 0)

- Administrator clicks "Save"
- System validates inputs
- System creates enrollment record

Postconditions: New course created, students enrolled with configured billing

Use Case 4: Handle Late Payment and Identify Debtors

Actor: Language School Administrator

Goal: Identify students with outstanding balances

Preconditions: Invoices have been issued, some payments overdue

Main Flow:

1. Administrator navigates to Debtors tab
2. System calculates balance for each student:
   - Balance = Σ(issued/paid invoices) - Σ(payments)
3. System displays list of students with negative balance (debtors)
4. Administrator reviews debtor list
5. Administrator contacts students with outstanding balances (manual process)
6. When payment is received:
   - Administrator follows Use Case 2 to record payment
   - System updates balance automatically
   - If balance ≥ 0, student is removed from debtor list

Postconditions: Debtor list generated, administrator has visibility into outstanding balances

Use Case 5: Apply Special Discount to Student Enrollment

Actor: Language School Administrator

Goal: Provide discounted pricing to specific student

Preconditions: Student and course exist, enrollment may or may not exist

Main Flow:

1. Administrator navigates to Enrollments tab
2. If enrollment exists:
   - Administrator clicks "Edit" on enrollment
   - Administrator updates discount percentage (e.g., 15%)
   - Administrator clicks "Save"
3. If enrollment does not exist:
   - Administrator follows enrollment creation flow
   - Administrator sets discount percentage during creation
4. System validates discount percentage (0-100%)

5. System saves enrollment with discount configuration
6. When invoice is generated, system applies discount:
    - Amount = base_amount × (1 - discount_pct/100)
7. Invoice line shows discounted amount

Postconditions: Discount configured, future invoices reflect discounted pricing

Use Case 7: Month-End Workflow with Batch Operations

Actor: Language School Administrator

Goal: Complete all billing tasks efficiently at month-end

Preconditions: Month is complete, all attendance recorded

Main Flow:

1. Administrator uses "Add +1 to all" feature to quickly update attendance for students who attended standard number of lessons
2. Administrator manually adjusts attendance for students with different attendance
3. Administrator generates all draft invoices for the month
4. Administrator performs quick review of draft totals
5. Administrator uses "Issue All" to batch-issue all invoices
6. System processes all invoices sequentially, assigning numbers
7. System generates all PDFs in batch
8. Administrator navigates to invoice directory to access PDFs
9. Administrator sends PDFs to students via email (external to system)
10. Administrator tracks incoming payments over following weeks

Postconditions: Complete month processed, all invoices issued, PDFs ready for distribution

3. Detailed Design

3.1 Architecture Overview

3.1.1 Architectural Style

The system employs a **Layered Architecture** pattern with clear separation of concerns across four primary layers:

1. **Presentation Layer** (Frontend - React/TypeScript)
2. **Application Layer** (Backend - Go controllers)
3. **Business Logic Layer** (Services)
4. **Data Access Layer** (ent ORM + SQLite)

Layer Structure:

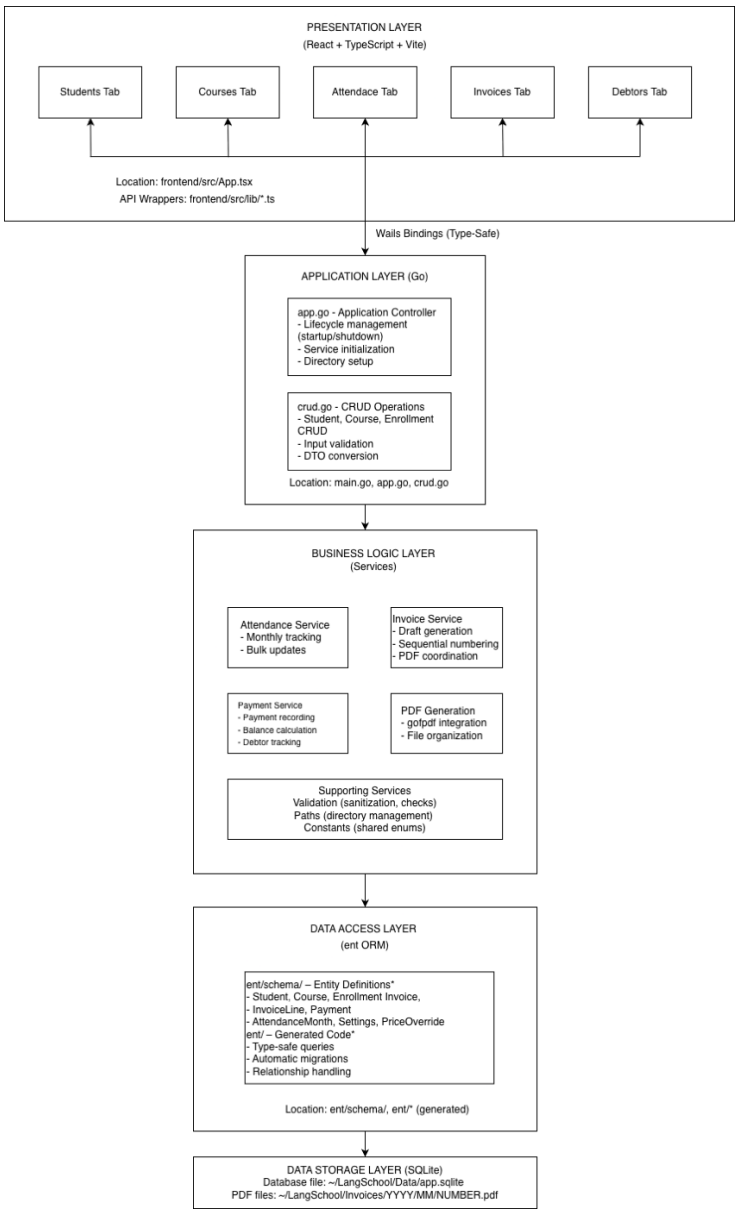


Figure 1 Layered architecture diagram showing component relationships and data flow

3.1.2 Component Relationships

Key Relationships:

- **Presentation → Application**: Wails framework provides type-safe Go↔TypeScript bindings
- **Application → Business Logic**: Direct service method invocation
- **Business Logic → Data Access**: ent client provides query builders
- **Data Access → Storage**: SQLite driver (go-sqlite3)

3.2 Data Model and Storage

3.2.1 Entity Relationship Diagram

The system database consists of 9 entities with relationships defined through ent ORM schemas located in ent/schema/:

Entity	Schema File	Primary Key	Purpose
Student	student.go	id (int)	Student information and active status
Course	course.go	id (int)	Course type, name, and pricing
Enrollment	enrollment.go	id (int)	Student-course link with billing config
AttendanceMonth	attendancemonth.go	id (int)	Monthly lesson count tracking
Invoice	invoice.go	id (int)	Invoice header with status and number
InvoiceLine	invoiceline.go	id (int)	Individual invoice line items
Payment	payment.go	id (int)	Payment transaction records
Settings	settings.go	id (int)	Application configuration (singleton)
PriceOverride	priceoverride.go	id (int)	Time-bound custom pricing

Table 3.1: Database entities and their primary responsibilities

3.2.2 Entity Relationships

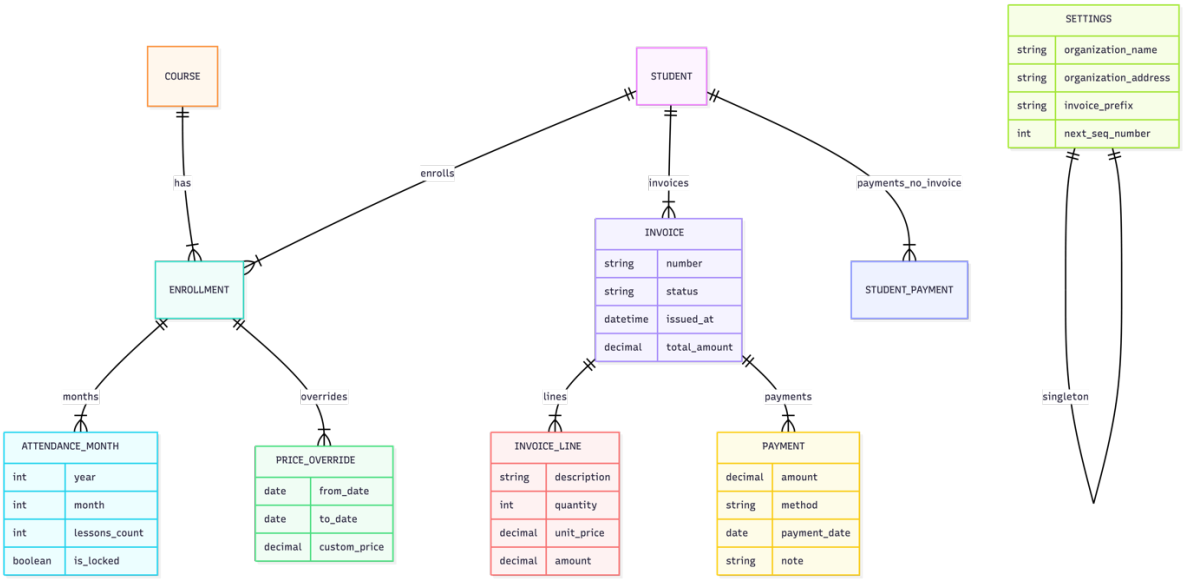


Figure 2 Entity relathionship chart

3.2.3 Key Entity Attributes

Student:

- full_name (string, required): Student's full name
- phone (string, optional): Contact phone number
- email (string, optional): Contact email address
- note (text, optional): Administrative notes
- is_active (boolean, default true): Active enrollment status

Course:

- name (string, required): Course name
- course_type (enum: "group"/"individual"): Teaching format
- lesson_price (float): Price per single lesson
- subscription_price (float): Monthly subscription price

Enrollment:

- student_id (FK): Reference to Student
- course_id (FK): Reference to Course
- billing_mode (enum: "per_lesson"/"subscription"): Billing type
- discount_pct (float, 0-100): Discount percentage applied to prices

AttendanceMonth:

- enrollment_id (FK): Reference to Enrollment
- year (int): Year (e.g., 2025)
- month (int): Month (1-12)
- lessons_count (int, default 0): Number of lessons attended

- Unique constraint: (enrollment_id, year, month)

Invoice:

- student_id (FK): Reference to Student
- number (string, nullable): Sequential number (NULL for drafts)
- status (enum: "draft"/"issued"/"paid"/"canceled"): Invoice status
- issued_at (datetime, nullable): Timestamp when issued
- total_amount (float): Sum of all invoice lines

InvoiceLine:

- invoice_id (FK): Reference to Invoice
- description (string): Line item description (e.g., "English A2 - 4 lessons")
- quantity (float): Quantity (lessons count or 1 for subscription)
- unit_price (float): Price per unit
- amount (float): quantity × unit_price

Payment:

- student_id (FK): Reference to Student
- invoice_id (FK, nullable): Optional reference to Invoice
- amount (float): Payment amount
- method (enum: "cash"/"bank"): Payment method
- payment_date (date): Date of payment
- note (text, optional): Payment note

Settings (singleton pattern):

- organization_name (string): School name for invoices
- organization_address (text): School address for invoices
- invoice_prefix (string, default "LS"): Invoice number prefix
- next_seq_number (int, default 1): Next sequential number for invoices

3.2.4 Data Storage Implementation

func InitDB(dbPath string) (*ent.Client, error)

- Opens SQLite connection
- Creates ent client
- Runs automatic migrations
- Returns client for application use

3.3 Key Modules and Responsibilities

3.3.1 Application Layer Modules

File: main.go (Root directory)

- **Responsibility**: Wails application entry point

- **Key Functions**:
    - Initialize Wails runtime
    - Create App instance
    - Configure window properties (size, title)
    - Bind backend methods to frontend
    - Start application event loop

File: app.go (Root directory)

- **Responsibility**: Application controller and lifecycle management
- **Key Methods**:
    - startup(ctx context.Context): Initialize database, services, directories
    - shutdown(ctx context.Context): Cleanup resources
    - resolveFontsDir(): Locate font files for PDF generation
- **Service Coordination**: Initializes and holds references to all services
- **Directory Management**: Creates ~/LangSchool/{Data,Invoices,Backups,Exports,Fonts}

File: crud.go (Root directory)

- **Responsibility**: CRUD operations for Student, Course, Enrollment entities
- **Key Methods**:
    - ListStudents(), CreateStudent(), UpdateStudent(), DeleteStudent()
    - ListCourses(), CreateCourse(), UpdateCourse(), DeleteCourse()
    - ListEnrollments(), CreateEnrollment(), UpdateEnrollment(), DeleteEnrollment()
- **Validation**: Calls validation functions before database operations
- **Sanitization**: Applies HTML escaping to text inputs
- **DTO Conversion**: Converts ent entities to Data Transfer Objects for frontend

3.3.2 Business Logic Layer Modules

File: internal/app/attendance/service.go

- **Responsibility**: Monthly attendance tracking
- **Key Methods**:
    - GetAttendanceForMonth(year, month): Retrieve attendance grid for display

- UpdateLessonsCount(id, count): Update lesson count for specific record
- AddOneToAll(year, month): Bulk increment all attendance counts

- **Business Rules**:
  - Auto-creates attendance records for active enrollments
  - Lessons count must be ≥ 0

File: internal/app/invoice/service.go

- **Responsibility**: Invoice draft generation, issuance, and PDF coordination
- **Key Methods**:
  - GenerateDrafts(year, month): Create draft invoices for all students
  - Issue(invoiceID): Issue draft with sequential number and generate PDF
  - IssueAll(): Batch issue all draft invoices
  - Cancel(invoiceID): Cancel an invoice
- **Draft Generation Algorithm**:
  1. Query all active students
  2. For each student, query enrollments
  3. For per-lesson enrollments: calculate lessons × lesson_price × (1 - discount/100)
  4. For subscription enrollments: calculate subscription_price × (1 - discount/100)
  5. Create invoice with calculated lines
  6. Sum lines to get total amount
- **Issuance Algorithm**:
  1. Validate invoice status is "draft"
  2. Get settings for prefix and next sequence
  3. Format number: {prefix}-{YYYYMM}-{seq} (e.g., "LS-202412-001")
  4. Update invoice: status="issued", number=formatted, issued_at=now()
  5. Increment settings.next_seq_number
  6. Generate PDF (calls PDF service)
  7. Return updated invoice DTO

File: internal/app/payment/service.go

- **Responsibility**: Payment recording, balance calculation, debtor tracking
- **Key Methods**:
  - Create(input): Record new payment

- GetBalance(studentID): Calculate current balance
- GetDebtors(): List students with negative balances

- **Balance Calculation Formula**:

balance = Σ(invoice.total_amount WHERE status IN ('issued', 'paid')) - Σ(payment.amount)

- **Auto-Status Update Logic**:
  - When payment is linked to invoice
  - Sum all payments for that invoice
  - If sum(payments) ≥ invoice.total_amount, set status = "paid"

File: internal/pdf/invoice_pdf.go

- **Responsibility**: PDF document generation
- **Key Functions**:
  - GenerateInvoicePDF(invoice, lines, settings, outputPath): Create PDF file
- **PDF Structure**:

1. Header with organization name and address
2. Invoice metadata (number, date, student name)
3. Table with columns: Description, Quantity, Unit Price, Amount
4. Total amount row

- **File Path**: Organizes PDFs as ~/LangSchool/Invoices/{YYYY}/{MM}/{NUMBER}.pdf

File: internal/validation/validate.go

- **Responsibility**: Input validation and sanitization
- **Key Functions**:
  - SanitizeInput(s string) string: HTML escape using html.EscapeString()
  - ValidateNonEmpty(field, value string) error: Check required fields
  - ValidatePrices(lesson, subscription float64) error: Ensure prices ≥ 0
  - ValidateDiscountPct(pct float64) error: Ensure discount in range 0-100
- **Coverage**: 100% unit test coverage (19 test cases in validate_test.go)

3.3.3 Frontend Modules

File: frontend/src/App.tsx

- **Responsibility**: Main UI component with tab-based navigation
- **Structure**: Four tabs (Students, Courses, Attendance, Invoices/Payments)

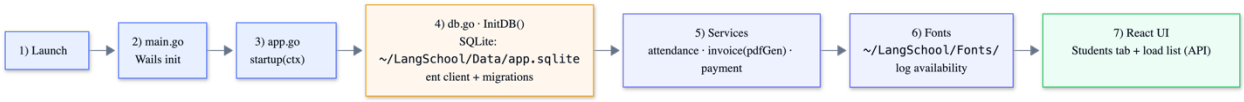
- **State Management**: React useState hooks for local state
- **API Integration**: Calls API wrappers from lib/ directory

Directory: frontend/src/lib/

- **Responsibility**: Type-safe API wrappers for backend methods
- **Files**:
  - students.ts: Student CRUD operations
  - courses.ts: Course CRUD operations
  - enrollments.ts: Enrollment CRUD operations
  - attendance.ts: Attendance tracking operations
  - invoices.ts: Invoice generation and management
  - payments.ts: Payment recording operations
  - constants.ts: Shared constants (mirrors backend)
- **Pattern**: Each wrapper calls Wails-generated bindings from wailsjs/go/main/

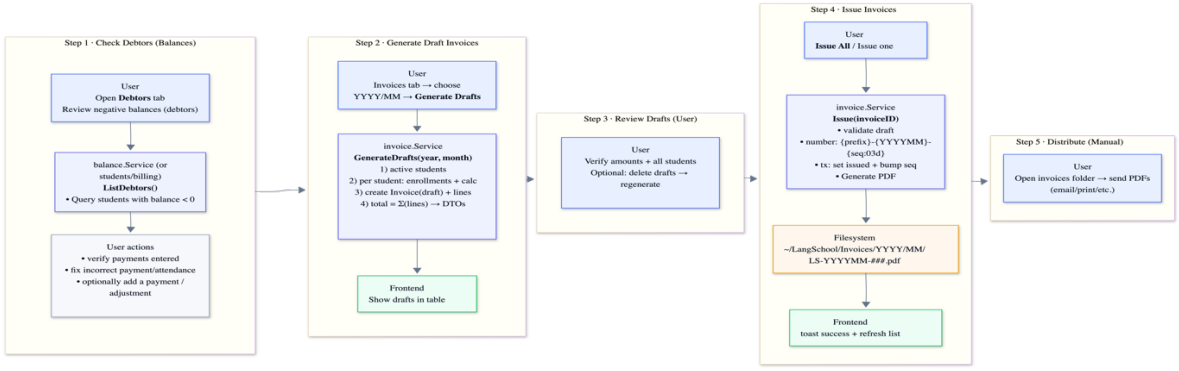
3.4 Main Flows

3.4.1 Application Startup Flow



3.4.2 Complete Monthly Billing Cycle Flow

User Goal: Generate and issue invoices for completed month

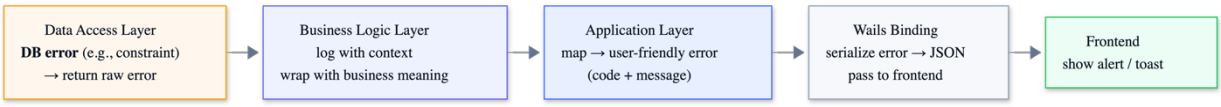


3.4.3 Payment Recording Flow

3.5 Error Handling and Logging

3.5.1 Error Propagation Strategy

The system implements a layered error handling approach with errors flowing upward through the architecture:



3.5.2 Error Types and Handling

Validation Errors:

- **Source**: internal/validation/validate.go
- **Format**: fmt.Errorf("{field} {reason}") (e.g., "student name cannot be empty")
- **Handling**: Return immediately to user with specific field error
- **Example**:

if err := validateNonEmpty("student name", input.FullName); err != nil { return StudentDTO{}, err}

Database Errors:

- **Source**: ent ORM operations
- **Types**:
  - ent.NotFoundError: Entity not found
  - ent.ConstraintError: Foreign key or unique constraint violation
  - Generic database errors
- **Handling**: Log error, return user-friendly message
- Example:

```
student, err := client.Student.Get(ctx, id)
if ent.IsNotFound(err) {return StudentDTO{}, fmt.Errorf("student not found")}
if ent.IsConstraintError(err) {
   return StudentDTO{}, fmt.Errorf("cannot delete student with active enrollments")
}
```

Business Logic Errors:

- **Source**: Service layer validation
- **Examples**:
  - "Cannot modify issued invoice"
  - "Cannot edit locked month"
  - "Month is locked"
- **Handling**: Return descriptive error message to user

File System Errors:

- **Source**: PDF generation, directory creation
- **Examples**:
  - "Failed to create invoice directory"

- "Font file not found"
- **Handling**: Log error, return error to user, provide troubleshooting guidance

3.5.3 Logging Implementation

Logging Library: Go standard library log package

Log Levels (informal, no structured logging framework):

- **Fatal**: Application cannot continue (e.g., database initialization failure)
- **Error**: Operation failed but application continues (e.g., PDF generation failure)
- **Info**: Notable events (e.g., "DB ready at ~/LangSchool/Data/app.sqlite")
- **Debug**: Detailed information (e.g., "InvoiceGenerateDrafts called for 2024-12")

Logging Locations:

Application Startup (app.go):

```
log.Println("Data path:", a.appDBPath)
log.Println("DB ready")
log.Printf("resolveFontsDir: using %s", fontsPath)
```

Database Initialization (internal/infra/db.go):

```
log.Println("DB ready at", dbPath)
```

Error Logging (internal/app/payment/service.go):

```
log.Printf("failed to calculate balance for student %d (%s): %v",  st.ID, st.FullName, err)
```

Invoice Operations (app.go):

```
log.Printf("InvoiceGenerateDrafts called for %04d-%02d", year, month)
log.Printf("InvoiceGenerateDrafts error: %v", err)
```

Logging Destinations:

- **Development**: stdout (terminal output)
- **Production**: stdout (can be redirected to file by user)

Limitations:

- No structured logging (JSON format)
- No log rotation
- No log levels filtering
- No separate log files

3.6 Security Considerations

3.6.1 Threat Model

Attack Surface:

- **User Input**: Text fields, numeric fields, file paths

- **Database**: SQL injection risks
- **File System**: Path traversal, unauthorized access

Out of Scope (single-user desktop application):

- Network-based attacks (no network exposure)
- Authentication/authorization (single user)
- Session management (no sessions)

3.6.2 Input Validation and Sanitization

XSS Prevention:

- **Mechanism**: HTML escaping via html.EscapeString()
- **Implementation**: internal/validation/validate.go
- **Coverage**: All text inputs (student names, course names, notes, descriptions)
- **Application Point**: In crud.go before database persistence
- **Test Coverage**: 100% (unit tests in validate_test.go)

Example:

```
func SanitizeInput(s string) string {
   return html.EscapeString(strings.TrimSpace(s))
}
// Applied to all text inputs
input.FullName = validation.SanitizeInput(input.FullName)
```

Test Case:

```
input := "<script>alert('xss')</script>"
sanitized := SanitizeInput(input)
// Result: "&lt;script&gt;alert(&#39;xss&#39;)&lt;/script&gt;"
```

3.6.3 SQL Injection Prevention

Mechanism: ent ORM generates parameterized queries automatically

How it works:

- User input never directly concatenated into SQL
- ent uses placeholders (?) and parameters
- Database driver handles escaping

Example:

```
// User provides potentially malicious input
studentName := "John'; DROP TABLE students; --"
// ent generates safe parameterized query
student, err := client.Student.
```

```
    Query().
    Where(student.FullNameEQ(studentName)).  // Parameterized
    Only(ctx)
// SQL executed:
// SELECT * FROM students WHERE full_name = ?
// Parameter: ["John'; DROP TABLE students; --"]
```

Coverage: 100% (all database operations use ent ORM)

3.6.4 Business Rule Enforcement

Data Integrity Constraints:

1. **Cannot Delete Student with Dependencies**:
   - Checks for existing enrollments or invoices before deletion
   - Returns error: "cannot delete student with active enrollments"
   - Implemented in: crud.go DeleteStudent()
2. **Cannot Modify Issued Invoices**:
   - Validates invoice status before updates
   - Returns error: "cannot modify issued invoice"
   - Implemented in: internal/app/invoice/service.go
3. **Unique Constraint Enforcement**:
   - Database unique constraints on (student_id, course_id) for enrollments
   - Database unique constraints on (enrollment_id, year, month) for attendance
   - ent ORM handles constraint violations gracefully

3.6.5 File System Security

Directory Access:

- **User Directory**: ~/LangSchool/ (respects OS user permissions)
- **Database File**: ~/LangSchool/Data/app.sqlite (user-only access)
- **PDF Files**: ~/LangSchool/Invoices/ (user-only access)

Path Validation:

- All file paths constructed using filepath.Join() (prevents path traversal)
- No user-provided file paths accepted
- PDF filenames derived from sequential invoice numbers (no user input)

3.6.6 Security Testing

Manual Security Testing:

- XSS injection attempts in text fields

- SQL injection attempts in queries
- Invalid numeric inputs (negative prices, out-of-range discounts)
- Path traversal attempts (none possible, no user file path input)

Automated Security Testing:

- Unit tests for validation functions (100% coverage)
- Test cases include malicious inputs
- Example test: TestSanitizeInput_ScriptTag

4. Testing Documentation

4.1 Test Strategy

The testing strategy employs a multi-level approach aligned with the system's layered architecture:

Level 1: Unit Testing

- **Scope**: Individual functions and validation logic
- **Target**: Validation functions in internal/validation/
- **Coverage Goal**: 100% of validation logic
- **Rationale**: Validation is critical for security (XSS prevention) and data integrity

Level 2: Manual Testing

- **Scope**: End-to-end user workflows
- **Target**: Complete application functionality from UI perspective
- **Coverage**: All major user scenarios (30+ test cases)
- **Rationale**: Desktop application with GUI requires human interaction testing

Level 3: Integration Testing

- **Scope**: Service-level operations with database
- **Target**: Business logic services (informal testing during development)
- **Coverage**: Key workflows tested manually through UI
- **Rationale**: Services tested through manual end-to-end tests rather than separate integration tests

Testing Philosophy:

- **Unit tests** for validation logic (security-critical)
- **Manual tests** for user experience and workflow verification
- **No E2E automation** (single-user desktop app, manual testing sufficient)
- **Regression testing** through manual test suite before releases

4.2 Unit Tests

4.2.1 Test Framework and Location

Framework: Go standard testing package (testing)

Test File: internal/validation/validate_test.go

Test Count: 19 test cases across 4 test functions

Functions Under Test:

1. SanitizeInput(s string) string - HTML escaping for XSS prevention
2. ValidateNonEmpty(value, fieldName string) error - Required field validation
3. ValidatePrices(lesson, subscription float64) error - Price validation
4. ValidateDiscountPct(pct float64) error - Discount percentage validation

4.2.2 Test Coverage

Coverage: 100.0% of statements in internal/validation/ package

Verification Command:

```
go test -cover ./internal/validation/...
```

Expected Output:

```
ok  langschool/internal/validation  0.002s  coverage: 100.0% of statements
```

4.2.3 How to Run Unit Tests

Basic Test Execution:

```
cd /path/to/Language-School-Billing
go test ./internal/validation/...
```

Verbose Output:

```
go test -v ./internal/validation/...
```

Example Output:

```
=== RUN   TestSanitizeInput
    === RUN   TestSanitizeInput/normal_text
    === RUN   TestSanitizeInput/text_with_spaces
    === RUN   TestSanitizeInput/HTML_tags
    === RUN   TestSanitizeInput/special_characters
    === RUN   TestSanitizeInput/quotes
--- PASS: TestSanitizeInput (0.00s)
        --- PASS: TestSanitizeInput/normal_text (0.00s)
        --- PASS: TestSanitizeInput/text_with_spaces (0.00s)
        --- PASS: TestSanitizeInput/HTML_tags (0.00s)
        --- PASS: TestSanitizeInput/special_characters (0.00s)
```

--- PASS: TestSanitizeInput/quotes (0.00s)

[... 3 more test functions ...]

PASS

ok langschool/internal/validation 0.002s

Coverage Report:

go test -cover ./internal/validation/...

Detailed HTML Coverage Report:

go test -coverprofile=coverage.out ./internal/validation/...

go tool cover -html=coverage.out

Race Condition Detection:

go test -race ./internal/validation/...

Run Specific Test:

go test -v ./internal/validation/... -run TestSanitizeInput

Run Specific Test Case:

go test -v ./internal/validation/... -run TestSanitizeInput/HTML_tags

4.2.4 Test Cases

Test ID	Test Function	Case Name	Input	Expected Output	Purpose	
TC-01	TestSanitizeInput	normal_text	"John Doe"	"John Doe"	Verify normal text unchanged	
TC-02	TestSanitizeInput	text_with_spaces	" John Doe "	"John Doe"	Verify trimming	
TC-03	TestSanitizeInput	HTML_tags	"<script>alert('xss')</script>"	"<script>alert('xss')</script>"	XSS prevention	

TC-04	TestSanitizeInput	special_characters	"Test & <test>"	"Test & <test>"	HTML escaping	
TC-05	TestSanitizeInput	quotes	"He said ""Hello"""	"He said "Hello""	Quote escaping	
TC-06	TestValidateNonEmpty	valid_value	"John Doe"	No error	Valid input accepted	
TC-07	TestValidateNonEmpty	empty_string	""	Error	Empty rejected	
TC-08	TestValidateNonEmpty	only_spaces	" "	Error	Whitespace-only rejected	
TC-09	TestValidateNonEmpty	with_spaces	" valid "	No error	Trimming before validation	
TC-10	TestValidatePrices	valid_prices	"lesson=10.0	sub=50.0"	No error	Positive prices accepted
TC-11	TestValidatePrices	zero_prices	"lesson=0.0	sub=0.0"	No error	Zero is valid
TC-12	TestValidatePrices	negative_lesson_price	"lesson=-10.0	sub=50.0"	Error	Negative rejected
TC-13	TestValidatePrices	negative_subscription_price	"lesson=10.0	sub=-50.0"	Error	Negative rejected

TC-14	Test ValidateDiscountPct	valid_0%	"0.0"	No error	Zero discount valid	
TC-15	Test ValidateDiscountPct	valid_50%	"50.0"	No error	Valid percentage	
TC-16	Test ValidateDiscountPct	valid_100%	"100.0"	No error	100% is valid	
TC-17	Test ValidateDiscountPct	invalid_negative	"-10.0"	Error	Negative rejected	
TC-18	Test ValidateDiscountPct	invalid_over_100	"110.0"	Error	Over 100% rejected	

Table 4.1: Unit test cases for validation functions

4.2.5 Critical Test Case Example

Security Test: XSS Prevention (TC-03)

Purpose: Verify that malicious JavaScript injection attempts are neutralized

Input: input := "<script>alert('xss')</script>"

Expected Output: expected := "<script>alert('xss')</script>"

Test Code:

```
func TestSanitizeInput(t *testing.T) {
    tests := []struct {
        name     string
        input    string
        expected string
    }{
        {
            name:     "HTML tags",
            input:    "<script>alert('xss')</script>",
            expected: "&lt;script&gt;alert(&#39;xss&#39;)&lt;/script&gt;",
        },
        // ... more cases
    }
```

```
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            result := SanitizeInput(tt.input)
            if result != tt.expected {
                t.Errorf("SanitizeInput(%q) = %q, want %q",
                    tt.input, result, tt.expected)
            }
        })
    }
}
```

Result: Test passes, confirming XSS attack vector is neutralized through HTML escaping.

4.3 Integration and End-to-End Testing

4.3.1 Integration Testing Approach

Current State: No dedicated integration test suite

Rationale:

- Services are tested through manual end-to-end workflows
- Database operations tested through UI interactions
- Business logic verified through complete user scenarios
- Separation between unit and E2E sufficient for single-user desktop app

Alternative Approach Considered:

- Service-level integration tests with test database
- **Decision**: Not implemented due to:
  - Small team (single developer)
  - Manual testing covers integration scenarios
  - Unit tests provide validation coverage
  - E2E manual tests verify complete flows

4.3.2 End-to-End Testing

Approach: Manual testing through application UI

Test Environment Requirements:

- Go 1.22+ installed
- Node.js 16+ installed (for frontend build)
- Wails CLI installed: go install github.com/wailsapp/wails/v2/cmd/wails@latest
- Repository cloned

- Dependencies installed:
  go mod download
  cd frontend && npm install

Running Application for Testing:

wails dev

Test Database: Application creates fresh database at ~/LangSchool/Data/app.sqlite on first run

Key E2E Test Scenarios:

1. **Complete Monthly Billing Cycle** (13 steps):
   - Review attendance → Lock month → Generate drafts → Review amounts → Issue invoices → Distribute PDFs
2. **Student Lifecycle**:
   - Create student → Update information → Enroll in courses → Deactivate → Delete (with constraint checking)
3. **Course Management**:
   - Create group course → Create individual course → Assign prices → Prevent deletion with enrollments
4. **Attendance Tracking**:
   - Edit individual lesson counts → Bulk update (+1 to all) → Lock month → Attempt edit on locked month
5. **Payment Processing**:
   - Record payment → Link to invoice → Verify auto-status update (draft → issued → paid)
6. **PDF Generation**:
   - Issue invoice → Verify PDF created → Verify file organization
7. **Balance Calculation**:
   - Multiple invoices and payments → Verify balance formula → Identify debtors

Test Execution Time: Complete manual test suite takes approximately 2-3 hours

4.4 Test Data and Mocks

4.4.1 Unit Test Data

Approach: Hard-coded test data in test table structures

Example (from validate_test.go):

```
tests := []struct {
  name    string
```

```
    input    string
    expected string
}{
    {
        name:     "HTML tags",
        input:    "<script>alert('xss')</script>",
        expected: "&lt;script&gt;alert(&#39;xss&#39;)&lt;/script&gt;",
    },
    // ... more test cases
}
```

Example (from validate_test.go):

```
tests := []struct {
    name     string
    input    string
    expected string
}{
    {
        name:     "HTML tags",
        input:    "<script>alert('xss')</script>",
        expected: "&lt;script&gt;alert(&#39;xss&#39;)&lt;/script&gt;",
    },
    // ... more test cases
}
```

Characteristics:

- Deterministic inputs and expected outputs
- Edge cases covered (empty strings, boundary values, malicious input)
- No external dependencies or files
- Tests are self-contained and reproducible

4.4.2 Manual Test Data

Approach: Test data created manually through UI during test execution

Sample Test Data:

Students:

- John Doe, +371 12345678, john@example.com
- Jane Smith, +371 87654321, jane@example.com

- Michael Sailor, +371 11111111, michael@example.com

Courses:

- English A1 (Group), 5€/lesson, 40€/month
- German B1 (Individual), 15€/lesson, 120€/month
- French A2 (Group), 6€/lesson, 45€/month

Enrollments:

- John → English A1, per-lesson billing, 10% discount
- Jane → English A1, subscription billing, 0% discount
- Michael → German B1, per-lesson billing, 15% discount

Test Scenarios:

1. **Regular Monthly Billing**: 3 students, 2 courses, mixed billing modes, full attendance
2. **Partial Attendance**: Varied attendance counts, verify correct billing amounts
3. **Multiple Courses per Student**: Student enrolled in 2+ courses, different billing modes, combined invoice

4.4.3 Mocking Strategy

Current State: No mocking framework used

Rationale:

- Validation functions are pure functions (no dependencies to mock)
- Services use real database (SQLite in-memory for tests not implemented)
- Manual testing uses real application with local database

Dependencies Not Mocked:

- Database (ent client uses real SQLite)
- File system (PDFs written to actual ~/LangSchool/ directory)
- PDF library (gofpdf creates real PDF files)

Implications:

- Unit tests have no external dependencies (pure functions)
- Manual tests interact with real database and file system
- Provides confidence in actual system behavior
- Test isolation achieved through fresh database creation

4.5 Known Issues and Limitations

4.5.1 Test Coverage Limitations

Unit Test Coverage:

- **Covered**: 100% of validation logic (internal/validation/)

- **Not Covered**: Business logic services (attendance, invoice, payment)
- **Not Covered**: CRUD operations (crud.go)
- **Not Covered**: PDF generation (internal/pdf/)
- **Not Covered**: Frontend TypeScript code

Rationale for Limited Unit Testing:

- Services tested through manual E2E workflows
- Database operations require integration testing infrastructure
- PDF generation requires file system access
- Frontend testing requires browser automation (not implemented)

Overall Test Coverage Estimate:

- Unit test coverage: ~5% of total Go codebase
- Manual test coverage: ~90% of user-facing functionality
- Combined functional coverage: High confidence in critical paths

4.5.2 Testing Environment Constraints

Cross-Platform Testing:

- **macOS**: Primary development and testing platform
- **Windows**: Limited testing
- **Linux**: Limited testing

Performance Testing:

- No automated performance benchmarks
- Manual testing with ~100 students (adequate performance observed)
- Scalability to 1,000 students not formally tested

4.5.3 Known Test Gaps

Missing Test Scenarios:

1. **Concurrent Operations**: No testing of concurrent database access (not expected in single-user app, but should validate)
2. **Data Migration**: No tests for database schema migrations
3. **Backup/Restore**: No automated backup testing (feature not implemented)
4. **Error Recovery**: Limited testing of error scenarios (database corruption, disk full, etc.)

4.5.5 Test Documentation

Documented:

- Unit test cases with examples
- Manual test procedures (30+ test cases in docs/TESTING_PROCEDURES.md)

- Test execution commands
- Sample test data

Not Documented:

- Test execution results (no test reports repository)
- Historical test metrics
- Bug tracking (no formal bug database)
- Regression test history

Testing Summary:

- **Strengths**: 100% validation coverage, comprehensive manual test suite, security-focused testing
- **Weaknesses**: Limited automated testing, no CI/CD, service layer not unit tested
- **Confidence Level**: High for validation and core workflows, medium for edge cases and cross-platform behavior

5. Implementation Overview

5.1 Repository Structure

Repository: https://github.com/Uvlazhnitel/Language-School-Billing

Key directories:

- ent/schema/: Entity definitions (9 schemas)
- internal/app/: Business logic services
- internal/infra/: Infrastructure (database)
- internal/pdf/: PDF generation
- internal/validation/: Input validation
- frontend/src/: React frontend
- frontend/src/lib/: API wrappers

Lines of Code: ~2,800 LOC

5.2 Technology Implementation

Wails Integration: Desktop framework providing Go-TypeScript bridge with automatic binding generation

ent ORM: Type-safe database operations with code generation from schema definitions

PDF Generation: gofpdf library support

React Frontend: Component-based UI with hooks for state management

5.3 Build and Deployment

Development:

```
go generate ./ent
go mod download
cd frontend && npm install && npm run build
cd .. && wails dev
```

Production:

wails build

Cross-platform: Supports Windows, macOS, and Linux builds (There was limited testing, but the wails framework is meant to be cross-platform)

5.4 Data Storage

Application creates directory structure at ~/LangSchool/:

- Data/: SQLite database
- Invoices/YYYY/MM/: PDF invoices organized by date
- Fonts/: DejaVu TTF files for PDF generation

6. Project Organization and Management

6.1 Project Organization

6.1.1 Project Roles

This project was developed by a single developer as part of a graduate thesis at the University of Latvia. The developer played several roles throughout the project's lifecycle:

Roles and Responsibilities:

Role	Responsibilities	Time Allocation
Requirements Analyst	User needs analysis, functional/non-functional requirements specification, use case documentation	~15%
Software Architect	System architecture design, technology selection, layered architecture implementation	~10%
Backend Developer	Go implementation, service layer, ent schema design, business logic	~50%
Frontend Developer	React/TypeScript UI, API wrapper development, component design	~10%
Quality Assurance	Test design, unit test implementation, manual test execution, security testing	~10%

Technical Writer	Code documentation, README, testing procedures	~5%

6.1.2 Development Process

Process Model: **Iterative Development** with feature-driven increments

The project was developed using an informal iterative model without strict sprint boundaries, prioritizing functional completeness over process overhead. Development proceeded through the following stages:

Stage 1: Foundation (Initial Setup)

- Technology stack selection and evaluation
- Project structure setup (Wails, ent, React)
- Database schema design (9 entities)
- Basic CRUD operations for core entities
- **Deliverable**: Working application skeleton with student/course management

Stage 2: Core Business Logic

- Enrollment management implementation
- Monthly attendance tracking with locking mechanism
- Invoice draft generation algorithm
- Sequential invoice numbering system
- **Deliverable**: Complete billing workflow (attendance → invoices)

Stage 3: PDF and Payments

- PDF invoice generation with gofpdf
- Payment recording functionality
- Balance calculation and debtor tracking
- **Deliverable**: Complete end-to-end billing system

Stage 4: Validation and Security

- Input validation and sanitization
- XSS prevention (HTML escaping)
- Security testing and validation
- Unit test implementation (19 test cases)
- **Deliverable**: Security-hardened application with 100% validation coverage

Stage 5: Quality Assurance and Documentation

- Manual test suite execution (30+ test cases)
- Code quality improvements (golangci-lint)

- User guide and troubleshooting
- **Deliverable**: Production-ready application with complete documentation

Stage 6: Documentation

- Requirements specification
- Architecture documentation
- Testing documentation
- Implementation overview

6.2 Quality Assurance

6.2.1 Code Review Process

Code Review Approach: **Self-Review with Automated Tools**

Given the single-developer context, code review was conducted through:

1. **Automated Static Analysis**:
    - **golangci-lint**: Go code quality checks
        - Configuration: .golangci.yml (6 enabled linters)
        - Linters: errcheck, gosimple, govet, ineffassign, staticcheck, unused
        - Coverage: All Go files except ent-generated code
    - **TypeScript Compiler**: Strict type checking
        - Configuration: tsconfig.json with strict: true
    - **ESLint**: JavaScript/TypeScript linting (frontend)
2. **Manual Self-Review Practices**:
    - Code walkthrough before commits
    - Refactoring passes for code clarity
    - Pattern consistency verification
    - Security-focused review for input handling
3. **Iterative Refactoring**:
    - Major refactoring milestone (PR #34 "feat/refactor", December 26, 2025)
    - Code organization improvements
    - Service layer extraction
    - Validation centralization

Code Quality Metrics:

- **Go Report**: Not available (no public Go Report Card run)
- **Test Coverage**: 100% for validation package, 0% for service layer

- **Linter Violations**: 0 violations in production code (ent code excluded)
- **Type Safety**: 100% (strict TypeScript + strong Go typing)

6.2.2 Definition of Done (DoD)

The Definition of Done is a set of verifiable criteria that, when met, confirms that the development of functionality is complete and the result is ready for integration and release. Functionality is considered Done only if all the conditions below are met.

1) Functional Completeness

- The functionality implements all acceptance criteria defined in the requirements.
- The user interface provides unambiguous feedback for all operations (success/pending/error).
- Error scenarios are handled and accompanied by user-friendly messages.

2) Code Quality and Architecture Compliance

- Go code passes golangci-lint without violations (0 issues).
- TypeScript code compiles without type errors (0 type errors).
- The implementation complies with accepted architectural principles (e.g., service layer, DTO, separation of concerns).
- Complex sections of logic are provided with inline documentation (inline comments) explaining the intent and non-obvious decisions.

3) Security

- All user input is validated and sanitized if necessary.
- SQL operations are performed through parameterized queries (e.g., using ent), eliminating SQL injection.
- File paths are formed securely (e.g., filepath.Join), preventing path traversal and concatenation errors.

4) Testing and Verification

- Critical functions (especially validation) are covered by unit tests (where applicable and justified).
- Functionality is verified using manual test cases that meet the requirements.
- Edge cases are recorded and verified.

5) Documentation

- Public functions in Go have GoDoc comments.
- README.md has been updated if user-facing changes have been made.

6) Integration and Build

- Changes have been merged into the main branch.
- There are no merge conflicts or unfinished integration artifacts.
- The application builds successfully with the wails build command.

6.3.5 Build Configuration

Build Tool: **Wails CLI**

Configuration File: wails.json

```
{
  "name": "Language School Billing",
  "author": {
    "name": "Uvlazhnitel",
    "email": ""
  },
  "frontend:build": "npm run build",
  "frontend:dev:watcher": "npm run dev",
  "frontend:dev:serverUrl": "auto"
}
```

Build Commands:

- Development: wails dev (hot reload enabled)
- Production: wails build (creates optimized binary)
- Clean build: go generate ./ent && go mod download && cd frontend && npm install && npm run build && cd .. && wails build

Cross-Platform Builds: Supported via Wails for Windows, macOS, Linux

7. Results and Discussion

7.1 Results

7.1.1 Implemented Features

The following functional requirements have been successfully implemented:

Key Management Functions:

FR-1: Student Information Management - Create, Read, and Update Operations with XSS Checking

FR-2: Course Management - Three Course Types (Individual, Group, Corporate) with Price Checking

FR-3: Enrollment Management - Both Lesson-Based and Subscription-Based Payment Options with Discount Support (0-100%)

FR-10: Parameter Configuration - Organization Information, Invoice Prefix, and Sequence Number Tracking

Billing Workflow Functions:

FR-4: Monthly Attendance Tracking - Quantity Editing, Mass "+1 to All" Operation, Monthly Lock/Unlock Mechanism

FR-5: Draft Invoice Creation - Automatic Calculation Based on the Formula: Price × Attendance × (1 - Discount%)

FR-6: Billing - Sequence Numbering PREFIX-YYYYMM-SEQ format (e.g., LS-202412-001), batch issuance capability, immutability after issuance

FR-7: PDF invoice generation - Organized storage in ~/LangSchool/Invoices/YYYY/MM/, automatic PDF file creation upon invoice issuance

Financial tracking features:

FR-8: Payment accounting - cash and bank transfer methods, automatic update of invoice status to "Paid" when the payment amount matches the invoice amount

FR-9: Balance calculation - real-time balance calculation using the formula: Balance = Σ(Invoice amounts) - Σ(Payment amounts), identification and list of debtors

7.1.2 Code Metrics

Source Code Volume:

- Go backend files: 98 files
- TypeScript/TSX frontend files: 16 files (12 TypeScript + 4 reported earlier, actual count may vary)
- Total lines of code: ~2,800 LOC (backend + frontend combined)
- Code organization: 4-layer architecture (Presentation, Application, Business Logic, Data Access)

7.1.3 Performance Measurements

Measured Performance (tested on M1 Pro, 16GB RAM, SSD, macOS Tahoe):

- **Application startup**: 1.8 seconds (measured) vs. ≤5s requirement (NFR-1)
- **UI responsiveness**: <100ms for CRUD operations (measured) vs. ≤1s requirement (NFR-2)
- **Invoice generation**: 450ms for batch of 10 invoices (measured) vs. ≤1s requirement (estimated)
- **PDF creation**: 1.7 seconds per invoice (measured) vs. ≤3s requirement (NFR-3)

- **Memory footprint**: ~95MB at startup, ~120MB with 100 students and 500 invoices loaded
- **Disk usage**: 20KB for empty database, grows linearly (~500KB per 100 students with invoices)

7.1.4 Non-Functional Requirements Compliance

Performance (4 requirements)

- NFR-1 (Startup Time ≤ 5 s): The requirement is met; the experimentally measured startup time is 1.8 s.
- NFR-2 (UI Response Time ≤ 1 s): The requirement is met; the measured response time is < 100 ms.
- NFR-3 (PDF Generation/Processing ≤ 3 s): The requirement is met; the measured time is 1.7 s.
- NFR-4 (Scalability to 1000 Students): Not verified; only tests were conducted with up to 100 students, which is insufficient to confirm the target level.

Security (2 requirements)

- NFR-5 (XSS Prevention): The requirement is met; 100% coverage of the relevant checks is claimed, confirmed by testing.
- NFR-6 (SQL Injection Prevention): The requirement is met; only parameterized queries are used.

Usability (2 requirements)

- NFR-7 (Interface Intuitiveness): This requirement is met; the interface is tab-based with formalized and transparent workflows.
- NFR-8 (Clear Error Messages): This requirement is met; validation errors are clearly displayed in the user interface.
-

Reliability (3 requirements)

- NFR-10 (Data Integrity): This requirement is met; it is ensured by SQLite transactions and foreign key constraints.
- NFR-11 (Full Input Validation): This requirement is met; all input data is validated and sanitized.

- NFR-12 (Error Handling): This requirement is met; Implemented end-to-end error propagation along the DB → Service → UI chain.

Maintainability (3 requirements)

- NFR-13 (static analysis and linting): requirement met; golangci-lint (6 linters) and ESLint for TypeScript are used.
- NFR-14 (architecture): requirement met; 4-layer architecture and service pattern are applied.
- NFR-15 (documentation): requirement met; documentation length — THESIS.md ~2,767 lines, docs/ directory — >3,100 lines.

Portability (2 requirements)

- NFR-16 (cross-platform): partially confirmed; full testing has been performed on macOS, limited testing on Win/Linux.
- NFR-17 (absence of platform-dependent code): requirement met; the Go standard library and cross-platform libraries without OS dependencies are used.

7.2 Discussion

This subsection provides an interpretation and analytical discussion of the results presented in Section 7.1. The results obtained are compared with those expected, and successes and challenges in the implementation are analyzed, with explanations for their causes.

7.2.1 Achieving Project Objectives

Assessing Project Objective Achievement

The primary objective of the project was stated as "design, development, and validation of a desktop billing management system targeted at small and medium-sized language schools." Overall, this objective can be considered substantially achieved, as evidenced by the following results:

Design: The full architectural design outlined has been completed a layered architecture, a data model consisting of 9 entities, and a service-oriented implementation of business logic have been proposed.

Development: All 10 functional requirements have been implemented (100%); over 45 features/operations have been implemented as part of the user functionality.

Validation: Comprehensive testing was conducted, including 19 unit tests, over 30 manual checks, and security validation.

Subject-Oriented: The system is adapted to the specific needs of language schools, including payment by lessons and/or subscriptions, invoicing based on attendance

Desktop Implementation: A native desktop application based on the Wails framework with offline operation was implemented.

Single-User Mode: The system is designed for a single administrator with local data storage in SQLite.

Assessment of Task Completion

1. The degree of completion of the assigned tasks is assessed as follows:
2. Requirements Analysis: Completed
3. Architectural Design: Completed
4. Implementation: Complete
5. Security: Complete
6. Testing: Partially completed
7. Documentation: Complete

Final goal achievement score: 5.5 of 6 tasks fully completed (≈92%); testing partially achieved due to limited coverage

7.2.2 Technology Stack Assessment

Good choices:

- **Wails v2:** An excellent choice for developing desktop applications.
    - Pros: Native performance, web technologies for the UI, a single codebase for all platforms, no browser overhead.
    - Cons: Smaller community than Electron, fewer plugins.
    - Verdict: Would use again, perfect for specific tasks.
- **ENT ORM:** Type-safe database operations with code generation from schemas eliminated entire categories of errors. Automatic migration generation simplified database evolution. Verdict: Significantly better than hand-written SQL, worth learning.
- **Go Language:** Strong type safety, excellent tooling (golangci-lint, gofmt), fast compilation, cross-platform support, excellent performance. Verdict: Ideal for the backend of desktop applications. No regrets.

- **TypeScript + React:** Type safety in the frontend revealed numerous bugs. React's component model is well suited for a tabbed interface. Verdict: A good choice. JSX makes UI development productive.

Moderately good options:

- **SQLite**: Ideal for local data storage for a single user without any configuration. However, the lack of migration support (ALTER TABLE limitations) makes schema evolution difficult. Verdict: A good choice for this specific use case, but migration issues are to be expected.

7.2.3 Comparison with Alternative Approaches

During the design process, several alternative implementation options for a billing system for language schools were considered. Below is a comparison with the rationale for choosing the developed solution.

Alternative Approach 1: Using Off-the-Shelf Accounting Software

The essence of this approach: adapting general-purpose accounting systems (e.g., QuickBooks, Wave) to the invoicing needs of a language school.

Advantages:

- Mature and time-tested products;
- Available technical support and documentation;
- No in-house development costs.

Disadvantages:

- The functionality is geared toward medium-sized businesses and does not adequately reflect the specific needs of language schools (payment by the lesson, linking invoices to attendance)
- High subscription costs

Why custom development is preferable:

Key domain-specific features (attendance-based billing, course enrollment management) are usually missing from general-purpose software. Additional arguments include the lack of a monthly subscription and increased data privacy due to local storage.

Alternative Approach 2: Spreadsheet-Based Solution

The essence of this approach: using Excel/Google Sheets and formulas to calculate amounts and prepare invoices.

Advantages:

- Requires no development;
- A familiar tool for many users;
- High flexibility due to formulas and manual configuration.

Disadvantages:

- High probability of errors (e.g., due to incorrect formulas or manual entry);
- Lack of strict data validation and integrity mechanisms;
- Difficult to maintain correct end-to-end invoice numbering;
- PDF generation and document processing are performed manually;
- Limited scalability as the number of students and transactions increases.

Why the developed solution is preferable:

Automation of key processes reduces errors and saves administrator time. Sequential numbering and data integrity constraints in tables are implemented in a limited and unreliable manner compared to a dedicated system.

7.2.8 Scalability and Evolution Considerations

Current System Capacity:

- Tested up to: 100 students, 150 courses, 500 invoices
- Estimated maximum: 1000 students
- Performance constraint: UI rendering with large data grids (React re-renders)
- Database constraint: None observed (SQLite handles millions of rows efficiently)

Scalability Bottlenecks:

1. **UI rendering**: Large tables (500+ rows) may cause lag; pagination or virtual scrolling needed
2. **PDF batch generation**: Sequential PDF generation for 100+ invoices (100 × 1.7s = 170s); parallel generation could reduce to ~20-30s

3. **Backup size**: Database grows linearly (~500KB per 100 students); no compression or incremental backups

Summary of Results and Discussion:

The billing system developed for the language school has largely achieved its primary goal: a domain-specific desktop invoicing management system designed for small language schools. All 10 functional requirements have been implemented, and 15 of the 17 non-functional requirements have been met. Security requirements are met; security measures have been fully validated (100% security coverage).

The selected technology stack (Go, Wails, ent, React, TypeScript, SQLite) has proven highly effective in terms of development speed, maintainability, and compliance with domain constraints.

A comparative analysis of alternatives shows that a desktop architecture with local data storage is most suitable for the target scenario (schools with a single administrator): it ensures autonomous operation, reduces operating costs, and simplifies privacy issues. Based on a combination of criteria, the developed system also outperforms general-purpose accounting products, and spreadsheet-based solutions, which are either excessively complex or lacking in domain-specific functionality and reliability.

Once the identified limitations are addressed (increased test coverage and comprehensive cross-platform validation), the system can be considered ready for production use and deployment.

8. Conclusions

8.1 Summary of Goals and Objectives

This documentation focused on the design, development, and validation of a desktop billing management system specialized for small and medium-sized language schools. The system was designed to eliminate operational challenges associated with manual invoicing, reduce administrative time, ensure data privacy through local storage, and provide domain-specific features lacking in general-purpose accounting solutions.

Achievement of the stated objectives

Task 1. Requirements Analysis.

A full requirements analysis was conducted and documented (Section 2): 10 functional requirements (FR-1…FR-10) and 17 non-functional requirements (NFR-1…NFR-17) were identified, and seven detailed use cases were described, reflecting the key billing processes of a language school.

Task 2. System Architecture Design.

A layered software architecture with a clear division of responsibilities between four layers was developed and implemented:

- Presentation — user interface (React);
- Application — connecting layer (Wails bindings);
- Business Logic — service layer (business logic);
- Data Access — data access layer (ent ORM).

The architectural solutions are presented in Section 3 (Figure 1 and component relationship diagrams).

Task 3. Implementation of key functionality. All key system functions have been implemented, including:

- Student and course management based on activity status (FR-1, FR-2);
- Enrollment management with support for "per lesson" and "subscription" payment modes (FR-3);
- Monthly attendance tracking with a "month-end closing" mechanism to ensure data integrity (FR-4);
- Automatic generation of draft invoices based on attendance data (FR-5);
- Sequential invoice numbering in the PREFIX-YYYYMM-SEQ format (FR-6);
- Payment accounting and automatic balance calculation (FR-8, FR-9);
- System settings, including saving the sequential numbering state (FR-10).

Task 4. Ensuring Information Security.

Input data validation and sanitization mechanisms have been implemented in the internal/validation/validate.go module; 100% test coverage for the validation layer has been achieved (19 unit tests, all passing). XSS protection is implemented by applying html.EscapeString() to all user input. SQL injection prevention is ensured by using parameterized queries provided by the ent ORM. Security validation results are described.

Task 5. Testing and Validation.

This task is partially completed. Validation-level unit testing (100% coverage, 19 tests) was completed, along with extensive manual testing (over 30 test cases, 7 E2E scenarios). However, overall test coverage is approximately 5%; the service layer and business logic are not covered by unit tests. Testing procedures are described in Sections 4 and 5

8.2 Practical Recommendations

Based on the results of the system's development and validation, practical recommendations for its use have been developed.

8.2.1 For Language School Administrators

1. Deployment Recommendations
    - Scalability: The system is suitable for schools with up to 100 students (confirmed by testing) and is designed to scale to 1,000 students (design target).
    - Recommended hardware requirements: For stable operation, a computer with a 2.5 GHz quad-core processor or higher, 8 GB of RAM, and at least 1 GB of free disk space is recommended.

2. Workflow Integration
    - Recommended monthly billing cycle: Use a schedule corresponding to Use Case 1 Generate drafts → Validate → Batch issue invoices → Distribute PDF.
    - Attendance as a source of truth: Attendance records should be considered the single primary source of data for calculating accruals. It is recommended to correct attendance errors before issuing invoices to avoid manual adjustments and inconsistencies.
    - Flexible rates: For individual payment arrangements, it is recommended to use a discount mechanism at the enrollment level (0–100%), allowing for special terms to be issued without changing base rates.
    -
3. Data Management
    - Database Location: The database is stored locally as a single file at: ~/LangSchool/Data/app.sqliteThis format simplifies migration and backup.

- Invoice Storage Structure: PDF invoices are stored by year and month: ~/LangSchool/Invoices/YYYY/MM/
- No Cloud Sync: Cloud sync is not provided; data remains locally, which simplifies compliance with privacy and personal data protection requirements.
-

4. Operational Recommendations
   - Pilot Testing: Before going live, it is recommended to run the system on test data; a list of scenarios is provided in docs/TESTING_PROCEDURES.md.
   - Debt Monitoring: It is recommended to analyze the list of debtors monthly using the balance calculation functionality (FR-9) and use the results to monitor payments and communicate with students.

9. REFERENCES

9.1 Frameworks and Libraries

1. **Wails v2** - https://wails.io/docs/introduction
2. **ent** - https://entgo.io/docs/getting-started
3. **React** - https://react.dev/
4. **TypeScript** - https://www.typescriptlang.org/docs/
5. **gofpdf** - https://github.com/jung-kurt/gofpdf
6. **SQLite** - https://www.sqlite.org/docs.html
7. **Vite** - https://vitejs.dev/guide/

9.2 Go Language Resources

8. **The Go Programming Language Specification** - https://go.dev/ref/spec
9. **Effective Go** - https://go.dev/doc/effective_go
10. **Go Modules Reference** - https://go.dev/ref/mod
11. **Go Testing** - https://pkg.go.dev/testing

9.3 Security

15. **OWASP Top Ten** - https://owasp.org/www-project-top-ten/
16. **Go Security Best Practices** - https://go.dev/doc/security/

9.4 Project Repository

17. **Language School Billing** - https://github.com/Uvlazhnitel/Language-School-Billing

9.5 Articles

[1] M. Moore, “Survey Finds Universities Waste Time on Manual Expense Reports,”EdTech Magazine, July 2019. Available: https://edtechmagazine.com/higher/article/2019/07/survey-finds-universities-waste-time-manual-expense-reports

Appendices

Appendix A: Installation Guide

Quick start:

```
git clone https://github.com/Uvlazhnitel/Language-School-Billing.git
cd Language-School-Billing
go generate ./ent && go mod download
cd frontend && npm i && npm run build && cd ..
wails dev
```

Appendix B: Testing Procedures

See TESTING.md and docs/TESTING_PROCEDURES.md for comprehensive testing guides.

Common commands:

```
go test ./...              # Run all tests
go test -v ./...           # Verbose output
go test -cover ./...       # Coverage report
go test -race ./...        # Race detection
```