Student Grades Management System

Functional Architecture and Developer Notes

WinForms UI → Services → SQLite

Admins & Viewers — Validation — Auth — Stats

Contents

1 Overview

The Student Grades Management System is a WinForms application built in F# that manages students, grades, and role-based access. It uses a thin UI over pure business logic and a SQLite database, keeping validation and error handling centralized. Two primary personas are supported:

- **Admin**: Full CRUD on students and grades, can inspect database tables, and view statistics.
- **Viewer**: Read-only access to student lists, per-student details, and class statistics.

2 Architecture

The solution separates concerns across domain types, validation/error utilities, business services, and WinForms UI. Figure 1 illustrates the runtime relationships.

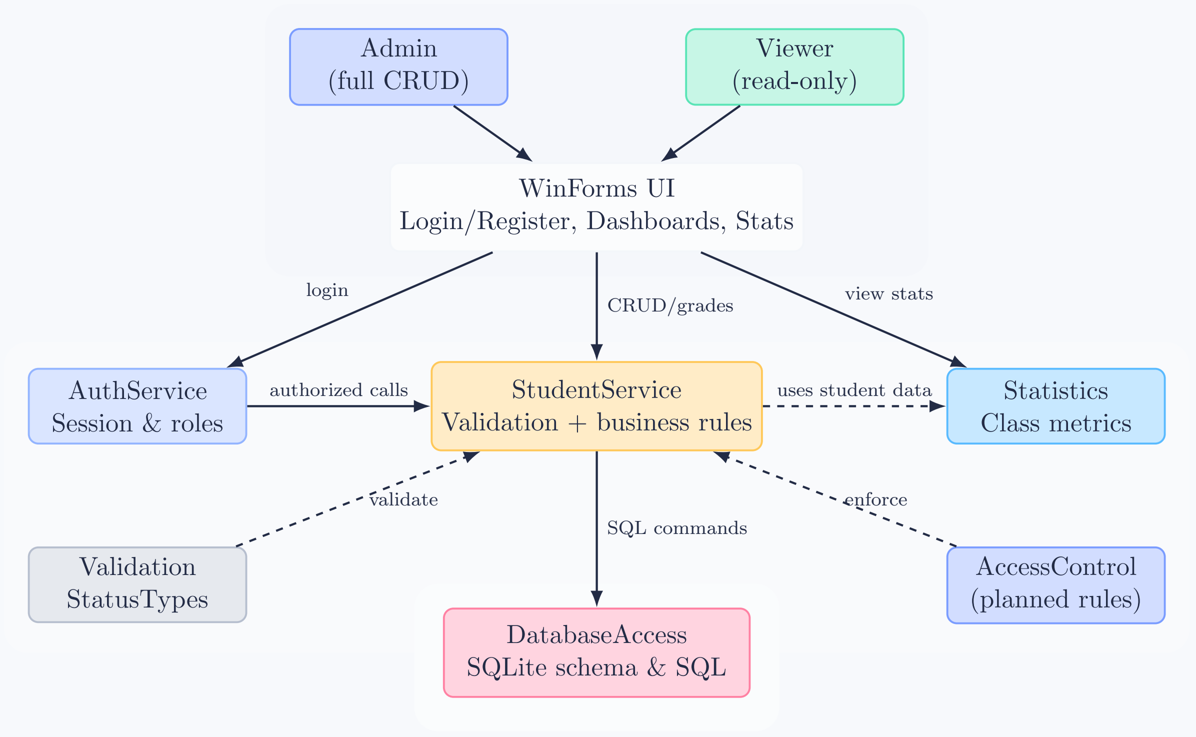

Figure 1: Block diagram: personas drive the UI; services mediate validation/auth/statistics before persisting to SQLite. Dashed arrows are supportive flows.

2.1 End-to-End Flow Diagram

Figure 2 walks through the primary user journeys and data paths, from authentication to CRUD/statistics and persistence.

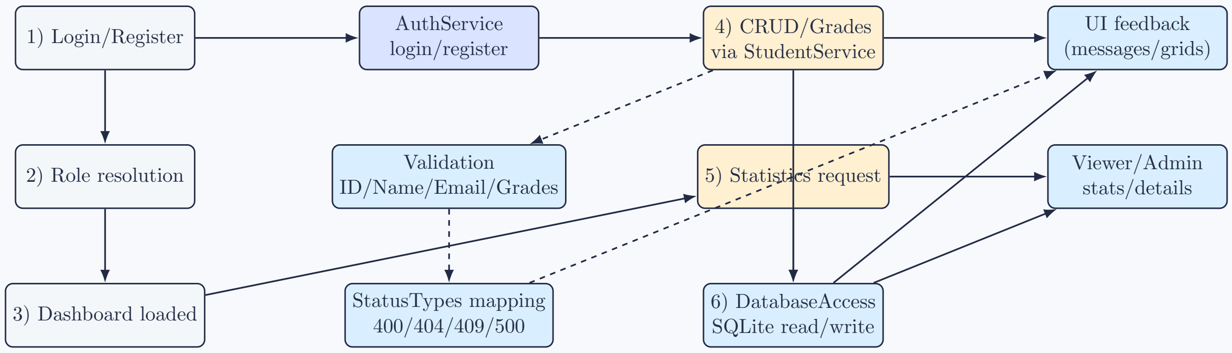
1) Login/Register
AuthService login/register
4) CRUD/Grades via StudentService
UI feedback (messages/grids)
2) Role resolution
Validation ID/Name/Email/Grades
5) Statistics request
Viewer/Admin stats/details
3) Dashboard loaded
StatusTypes mapping 400/404/409/500
6) DatabaseAccess SQLite read/write

Figure 2: End-to-end flow: authentication leads to dashboards; actions pass through validation and status mapping, hit SQLite, and return feedback/results to the UI. Dashed arrows highlight validation/error propagation.

2.2 Activity Diagram

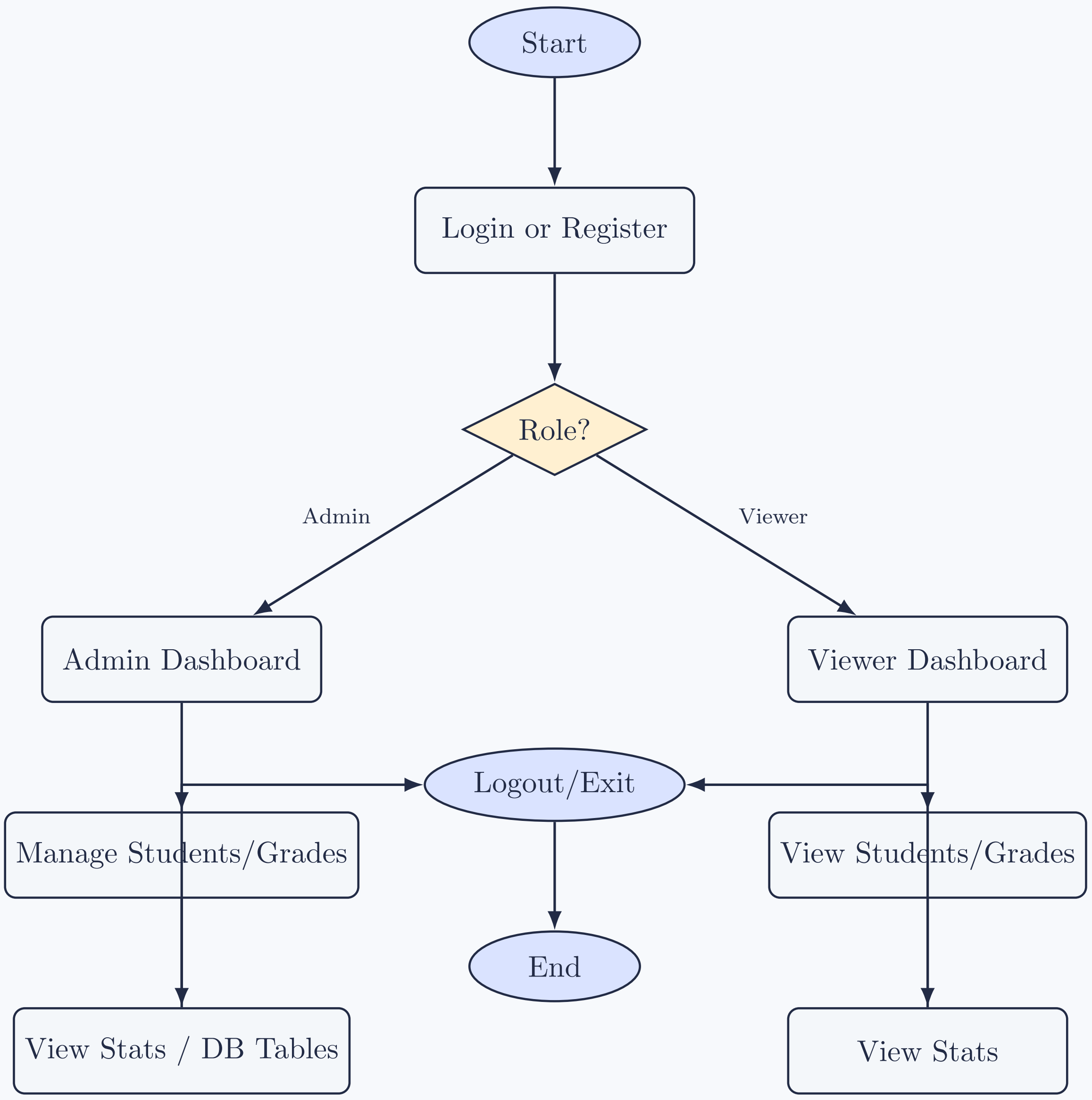

Figure 3: Activity diagram: users authenticate, branch by role to Admin or Viewer activities, then converge on logout/exit.

3 Codebase Layout

- **Domain.fs**: Core types for students, users, roles, and aggregate stats.
- **Validation.fs**: Central input validation (IDs, names, emails, grades) used by services before persistence.
- **StatusTypes.fs**: Uniform status codes and error discriminated unions for consistent UI messaging.

- **Statistics.fs**: Pure computations for per-student averages and class metrics; no I/O dependencies.
- **DatabaseAccess.fs**: SQLite schema creation and CRUD for students, grades, admins, and viewers (synchronous ADO calls).
- **StudentService.fs**: Business logic orchestrating validation and database access; translates validation errors to `StatusError`.
- **AuthService.fs**: Login, registration for viewers/admins, session state helpers, and admin seeding on startup.
- **AccessControl.fs**: Placeholder for fine-grained permission checks (currently documented but not enforced).
- **WinForms UI**: `LoginForm`, `RegisterForm`, `AdminDashboardForm`, `ViewerDashboardForm`, `StatisticsForm`, `DatabaseViewerForm` (buttons/dialogs wired to services).
- **Program.fs**: Entry point that initializes the database and launches the login form.
- **StudentManagement.Tests**: xUnit tests using a fake in-memory database to validate service logic.

4 Detailed Behavior

4.1 Authentication and Roles

- **Startup seed**: `DatabaseAccess.initializeDatabase()` creates tables and seeds an `admin/admin` account if none exists.
- **Login**: `AuthService.login` checks Admins table first, then Viewers; sets `currentUser` with `UserRole`.
- **Role helpers**: `isAdmin/isStudent/isViewer` gate UI routing; AccessControl is ready for future permission checks.
- **Registration**: `RegisterForm` calls `AuthService.registerViewer` (unique username enforced). `addNewAdmin` allows adding admins from code.

4.2 Admin Dashboard

- CRUD buttons surface `StudentService` calls; validation errors return typed `StatusError` mapped to HTTP-like codes for messages.
- Grade buttons manage per-subject scores; duplicate subjects are rejected; missing students/subjects return `NotFoundError`.

- Statistics button launches `StatisticsForm` using the default pass threshold of 75.
- Database tables viewer shows raw SQL views for auditing Students, Grades, Admins, and Viewers.

4.3 Viewer Dashboard

- Read-only list of students; per-ID view renders student info and grades in a grid.
- Statistics modal uses the same `Statistics.classStats` flow as Admin.
- Logout clears `AuthService.currentUser`.

4.4 Validation and Error Mapping

- **ID**: Must be positive and unique. **Name**: non-empty, min 2 chars. **Email**: non-empty, contains "@". **Grade**: subject non-empty, score 0–100.
- Validation issues return `ValidationError`; duplicates map to `ConflictError`; missing entities map to `NotFoundError`; database exceptions map to `ServerError`.
- UI displays the mapped status code (e.g., 400, 404, 409, 500) alongside the message.

4.5 Database Schema and Persistence

- **Students**: `Id` (PK, explicit), `Name`, `Email` (UNIQUE).
- **Grades**: `Id` (PK, auto), `StudentId` (FK with cascade delete), `Subject`, `Grade`; unique per (`StudentId`, `Subject`).
- **Admins**: `Id` (PK), `Name` (UNIQUE), `Password`, `Email`, `CreatedDate`; seeded admin uses plain text (replace with hashing for production).
- **Viewers**: `Id` (PK), `Name` (UNIQUE), `Password`, `Email` (UNIQUE), `CreatedDate`.
- All DB calls are synchronous via `Microsoft.Data.Sqlite`; errors propagate as `Result<_, string>` then wrap into `StatusError`.

4.6 Statistics Details

- `studentAverage`: returns `None` for students with no grades; otherwise averages all scores.
- `isPassing`: averages available grades and compares to a threshold (default 75); no grades means failing.
- `classStats`: aggregates count, highest/lowest averages (optionals), and pass rate (optional when no students).

4.7 UI Interaction Flows

- **LoginForm**: validates non-empty username/password, calls `AuthService.login`, routes to Admin or Viewer dashboards based on role.
- **RegisterForm**: validates non-empty fields, calls `AuthService.registerViewer`; on success, closes and returns to login.
- **AdminDashboardForm**: each button opens an InputBox, forwards to `StudentService` or `Statistics`, and shows message boxes with status codes.
- **ViewerDashboardForm**: renders student list in a grid; ID lookup opens grades grid and summary label; statistics opens a modal.
- **StatisticsForm**: simple read-only view of aggregate numbers with formatted optionals.
- **DatabaseViewerForm**: TabControl with four grids bound to SQL queries for transparent inspection.

4.8 Service API Summary

- **StudentService.createStudent**: validates ID/Name/Email; inserts student; returns `Result<string, StatusError>`.
- **StudentService.modifyStudent/removeStudent**: ensure existence, validate input, then update/delete.
- **StudentService.addStudentGrade**
  updateStudentGrade
  removeStudentGrade: validate subject/grade, ensure student and subject rules, then insert/update/delete grade rows.
- **AuthService.login**: check admins then viewers; sets mutable `currentUser`.
- **AuthService.registerStudent**
  registerViewer
  addNewAdmin
  removeAdmin: convenience wrappers around DB operations with duplicate checks.
- **Statistics.classStats**: pure aggregation over in-memory student list.

4.9 Operational Notes

- **Platform**: targets `net8.0-windows` with WinForms; requires Windows runtime for the UI.
- **Database**: SQLite file `students.db` in project root; delete with care, schema recreated on next launch (data lost).

- **Passwords**: stored in plain text in SQLite for admins/viewers; replace with hashing (e.g., PBKDF2/BCrypt) before production use.
- **Concurrency**: DB access is synchronous; if multi-user is needed, add transactions and concurrency checks.
- **Localization**: UI strings are English with some inline Arabic comments; centralize strings for translation if required.

4.10 Future Hardening

- Enforce `AccessControl.canPerform` before mutating actions; deny UI buttons for unauthorized roles.
- Add input sanitization against SQL special characters (parameterized queries already mitigate injection).
- Introduce logging (info/error) around DB calls and authentication attempts.
- Add subject-level statistics (per-subject averages, top performers) in `Statistics.fs`.
- Add integration tests using an ephemeral SQLite database to cover end-to-end flows with UI-less harnesses.

4.11 Testing Notes

- `StudentManagement.Tests` replaces `DatabaseAccess` with an in-memory fake module to avoid SQLite dependency.
- Coverage includes: student create/update/delete, duplicate detection, grade add/update/remove.
- Run with `dotnet test`; extend by adding role/authorization and statistics edge-case tests.

5 Domain Model

- **Student**: {Id:int; Name:string; Email:string; Grades: Map¡string, float¿}.
- **UserRole**: `Admin | StudentUser of int | Viewer of int`.
- **User**: {Id:int; Name:string; Password:string; Role:UserRole}.
- **ClassStats**: Aggregates for student count, highest/lowest averages, and pass rate.
- **StatusError**: Validation, not-found, conflict, or server errors with mappable HTTP-like codes.

6 Database Schema

SQLite database `students.db` is created at startup.

- **Students(Id PK, Name, Email UNIQUE)**: Base student records.
- **Grades(Id PK, StudentId FK, Subject, Grade, UNIQUE(StudentId, Subject))**: Per-subject scores.
- **Admins(Id PK, Name UNIQUE, Password, Email, CreatedDate)**: Admin credentials; seeded with `admin/admin`.
- **Viewers(Id PK, Name UNIQUE, Password, Email UNIQUE, CreatedDate)**: Read-only users.

7 Core Flows

7.1 Authentication and Session

1. User submits credentials in `LoginForm`.
2. `AuthService.login` queries admins, then viewers; on success it records the current user and role.
3. Admins are routed to `AdminDashboardForm`; viewers go to `ViewerDashboardForm`.

7.2 Student and Grade Management

1. UI collects input (IDs, names, emails, subjects, grades).
2. `StudentService` composes validation, existence checks, and database operations.
3. On success, operations return user-friendly messages;
   on failure, they return typed `StatusError` values.

7.3 Statistics

1. UI requests class metrics (default pass threshold 75).
2. `Statistics.classStats` computes highest/lowest averages and pass rate using current student data.
3. Results are rendered in `StatisticsForm`.

8 Validation and Error Handling

- **IDs**: Must be positive and unique.
- **Names**: Non-empty, at least two characters.
- **Emails**: Non-empty and must contain "@".
- **Grades**: Subject cannot be empty; grade range is 0–100.
- Errors are captured as `StatusError` and mapped to HTTP-like codes for consistent UI messaging.

9 WinForms UI Highlights

- **LoginForm**: Authentication and branching to dashboards; launches viewer registration dialog.
- **RegisterForm**: Collects viewer info and invokes `AuthService.registerViewer`.
- **AdminDashboardForm**: Buttons for CRUD, grade management, statistics, database table viewer, and logout.
- **ViewerDashboardForm**: Read-only student list, per-student grades, statistics, and logout.
- **DatabaseViewerForm**: Tabbed grids for Students, Grades, Admins, and Viewers (direct SQL views).
- **StatisticsForm**: Displays aggregate metrics in a simple layout.

10 Testing

The `StudentManagement.Tests` project uses xUnit with an in-memory fake database module to validate business rules:

- Creating, updating, and deleting students.
- Duplicate detection for students and grades.
- Grade add/update/remove success paths.

Run with `dotnet test` from the repository root.

11 Build and Run

1. Ensure the .NET 8 SDK is installed.
2. Build: `dotnet build StudentManagement.fsproj`.
3. Run: `dotnet run --project StudentManagement.fsproj`. The app initializes SQLite schema and opens the login window.
4. Tests: `dotnet test StudentManagement.Tests/StudentManagement.Tests.fsproj`.
5. LaTeX documentation: `pdflatex documentation.tex`.

12 Extensibility Notes

- Fill in `AccessControl.fs` to enforce per-role permissions before invoking service actions.
- Harden password storage (hashing) and input validation for production scenarios.
- Add pagination or search filters to student listings as the dataset grows.
- Expand statistics (standard deviation, per-subject averages) by extending `Statistics.fs`.
- Introduce dependency injection to swap `DatabaseAccess` for mockable abstractions in UI tests.