Student Grades

Management System

Functional-First F# Design

Functional, immutable, and JSON-backed.

Console-based student records, immutable workflows, JSON persistence, and role-aware access control.

Version 1.0
Built for clarity, testability, and clean separation of concerns.

Role Admin & Viewer **Access Control**
Persistence JSON **Immutable State**

1 Project Description

Student Grades Management System (F# – Functional Design) is a console-based application that manages student records and their grades using a purely functional design in F#. The system allows an administrator to add, edit, and delete student records, enter grades for multiple subjects, and generate useful reports such as per-student averages and overall class statistics.

The application implements simple role-based access control. Users can log in as either an **Admin** or an **Viewer**. Admins have full access to modify student data (CRUD operations), while Viewers are restricted to read-only access to student reports and statistics.

To support persistence, the system saves all student data to a JSON file and can reload it when the application starts. This makes it possible to keep data across multiple runs without using a full database.

The project emphasizes functional programming principles: immutable data structures, pure functions, clear type definitions (records and discriminated unions), and separation between pure business logic and side-effects (I/O, JSON file access, user input).

2 Design in F#

Core Data Model (Domain)

Everything is a value; the database is a list of students. Suggested types:

```
type Role =
    | Admin
    | Viewer

type Grade = {
    Subject : string
    Score   : float
}

type Student = {
    Id        : int
    Name      : string
    Grades    : Grade list
    IsPassing : bool
}

type ClassStats = {
    StudentCount : int
    HighestAvg   : float option
    LowestAvg    : float option
    PassRate     : float option
}

type AppState = {
    Students : Student list
}
```

Functional Design Principles

- Immutability: no in-place mutation; functions return new lists/records.
- Pure functions: signatures like `Student list -> Student list` or `Student list -> ClassStats`; no I/O.
- Side effects at the edges: only UI handles console I/O and JSON file access.

Implementing the Features

CRUD (Add, Edit, Delete)

```
module StudentCrud =

    let addStudent (newStudent : Student) (students : Student list) : Student list =
        newStudent :: students

    let updateStudent (updated : Student) (students : Student list) : Student list =
        students
        |> List.map (fun s -> if s.Id = updated.Id then updated else s)

    let deleteStudent (id : int) (students : Student list) : Student list =
        students
        |> List.filter (fun s -> s.Id <> id)
```

Grade Calculations

```
module GradeCalc =

    let averageScore (grades : Grade list) : float option =
        match grades with
        | [] -> None
        | _ ->
            let sum = grades |> List.sumBy (fun g -> g.Score)
            let count = grades |> List.length |> float
            Some (sum / count)

    let isPassing (passThreshold : float) (grades : Grade list) : bool =
        match averageScore grades with
        | None -> false
        | Some avg -> avg >= passThreshold

    let recalcStudentPassStatus passThreshold (student : Student) : Student =
        let passing = isPassing passThreshold student.Grades
        { student with IsPassing = passing }
```

Class Statistics

```
module Statistics =

    let studentAverage (s : Student) : float option =
        GradeCalc.averageScore s.Grades

    let classStats (passThreshold : float) (students : Student list) : ClassStats =
        let avgs =
            students
            |> List.choose (fun s -> studentAverage s)

        let highest =
            match avgs with
            | [] -> None
            | _  -> Some (List.max avgs)

        let lowest =
            match avgs with
            | [] -> None
            | _  -> Some (List.min avgs)

        let passCount =
            students
            |> List.filter (fun s -> GradeCalc.isPassing passThreshold s.Grades)
            |> List.length

        let total = students |> List.length

        let passRate =
            if total = 0 then None
            else Some (float passCount / float total)

        {
            StudentCount = total
            HighestAvg   = highest
            LowestAvg    = lowest
            PassRate     = passRate
        }
```

Role-Based Access Control

```
module AccessControl =

    type Action =
        | ViewStudents
        | ViewStats
        | AddStudent
        | EditStudent
        | DeleteStudent

    let canPerform (role : Role) (action : Action) : bool =
```

```
        match role, action with
        | Admin, _ -> true
        | Viewer, ViewStudents -> true
        | Viewer, ViewStats    -> true
        | Viewer, (AddStudent | EditStudent | DeleteStudent) -> false
```

JSON Save/Load (Persistence)

```
module Persistence =

    open System.IO
    open System.Text.Json

    let saveToFile (path : string) (state : AppState) =
        let options = JsonSerializerOptions(WriteIndented = true)
        let json = JsonSerializer.Serialize(state, options)
        File.WriteAllText(path, json)

    let loadFromFile (path : string) : AppState =
        if File.Exists(path) then
            let json = File.ReadAllText(path)
            JsonSerializer.Deserialize<AppState>(json)
        else
            { Students = [] }
```

UI – Main Program Loop (Console)

```
module Ui =

    let rec mainLoop (role : Role) (state : AppState) =
        printfn "1) View all students"
        printfn "2) View class statistics"
        if role = Admin then
            printfn "3) Add student"
            printfn "4) Edit student"
            printfn "5) Delete student"
        printfn "0) Save & Exit"

        let choice = System.Console.ReadLine()

        match choice, role with
        | "1", _ ->
            // display students
            mainLoop role state

        | "2", _ ->
            // compute and display stats
            mainLoop role state
```

```
        | "3", Admin ->
            // add student
            mainLoop role state

        | "4", Admin ->
            // edit student
            mainLoop role state

        | "5", Admin ->
            // delete student
            mainLoop role state

        | "0", _ ->
            Persistence.saveToFile "students.json" state

        | _ ->
            printfn "Invalid choice."
            mainLoop role state
```

3 Team Roles and Responsibilities

1. **Student Model Designer**: Owns `Domain.fs`; defines Role, Grade, Student, ClassStats, AppState; documents field names and types.
2. **CRUD Developer**: Owns `StudentCrud.fs`; implements add, update, delete, find; all pure list functions.
3. **Grade Calculation Developer**: Owns `GradeCalc.fs`; implements averageScore, isPassing, recalcStudentPassStatus; may add grade helpers.
4. **Statistician**: Owns `Statistics.fs`; implements studentAverage and classStats; may add top/failing helpers.
5. **Role Manager**: Owns `AccessControl.fs`; defines actions and permissions; may add simple login.
6. **Persistence Developer**: Owns `Persistence.fs`; wraps JSON/file logic; exposes saveToFile and loadFromFile.
7. **UI Developer**: Owns `Ui.fs` and `Program.fs`; builds menus and formatting; wires other modules.
8. **Documentation and Version Control Lead**: Creates folder layout, writes README and design notes, keeps Git hygiene.

4 Summary of Best Practices

- Start with types and agree on signatures early.
- Keep logic pure; isolate side effects to UI and persistence modules.
- Use pattern matching and options for missing data cases.
- Keep JSON and console handling isolated; treat other modules as a pure library.
- Use small Git branches and frequent commits per feature.