SOLIDS PROJECT

-Arvind Venkat Ramanan

AIM:

To develop a Backend Engine by following the principles of SOLIDS.

PROPOSED METHOD - SOLID Principles Implementation:

- **Single Responsibility Principle (SRP):**
  - Model Classes: Student, ContactInfo, Marks, and Report each have a single responsibility: representing student data. No one class should be responsible for multiple, unrelated aspects (e.g., data access and report generation).
  - Repositories: StudentRepository is solely responsible for data access (CRUD operations).
  - Services: StudentServices handles business logic related to students (e.g., calculating GPA, generating reports).
  - Controllers: StudentController handles API requests and responses.

- **Open/Closed Principle (OCP):**
  - Interfaces: Use interfaces (StudentRepository, StudentServices) to define contracts. Implementations can be changed without modifying the code that uses the interfaces. For example, you could switch from an in-memory database to a relational database by providing a new implementation of StudentRepository.
  - Dependency Injection: Use dependency injection to inject dependencies (e.g., StudentRepository into StudentServices). This makes it easy to switch implementations without code changes.

- **Liskov Substitution Principle (LSP):**
  - If you have different types of reports, any function that uses the Report type should be able to use any of its subtypes without needing to know the specific subtype.

- **Interface Segregation Principle (ISP):**
  - Avoid large, monolithic interfaces. If different parts of your system need only a subset of methods, create smaller, more specific interfaces. |

- **Dependency Inversion Principle (DIP):**
  - High-level modules (e.g., StudentServices) should not depend on low-level modules (e.g., StudentRepository). Both should depend on abstractions (interfaces). This is achieved through interfaces and dependency injection.

METHOD OFAPPROACH

I have created the data models for my student database using C# classes. Each class represents a specific aspect of student information, promoting data organization and modularity. These models are designed to be used with an Object-Relational Mapper (ORM) like Entity Framework Core to interact with a relational database.

Specifically, I have defined the following models:

- **StudentInfoModel:**
  This model stores core student details, including a unique StudentId (using a Guid for a globally unique identifier), StudentName, Class_Section, DOB (Date of Birth), and BloodGroup. Data annotations are used to specify data types, display names, and validation rules (e.g., [Key], [Required], [DisplayFormat], [DataType]).
- **StudentContactInfoModel:**
  This model holds student contact information, including the StudentId (linking it to the StudentInfoModel), ContactName, ContactNumber, and ContactEmailId. Again, data annotations are used for validation and display.
- **StudentMarksModel:**
  This model stores student marks for different subjects. It includes the StudentId, Class_Section, and then fields for each subject (Subject1, Subject2, Subject3), their corresponding percentages (Subject1_percentage, etc.), and grades (Subject1_Grade, etc.).
- **StudentReportModel:**
  This model contains information relevant to student reports, such as the StudentId, AadharCardNumber, and boolean flags indicating whether term fees have been paid (Term1_Fees, Term2_Fees).

The **StudentId** acts as a primary key in each model and will be used as a foreign key in the database to establish relationships between the tables, ensuring data consistency and integrity. This design allows for efficient storage and retrieval of student data, as well as the ability to easily expand the database with additional information in the future.