**Designing MediBuddy**

Let’s design a system where a hospital management able to maintain essential information of doctors, patients and respective specializations.

**What is MediBuddy?**

MediBuddy is a web application, that helps hospital management to view, add, edit and delete user information. In modern world, rather than storing registered information of doctors and patients in hardcopy notebooks, this system allows to maintain user records in an efficient manner.

Here we plan to design a simpler Hospital Management System, which will perform basic CRUD operations on registered users.

**Requirements and Goals of the System**

We will focus on the following set of requirements while designing the MediBuddy:

**Functional Requirements**

1. User should be able to view, add, update and delete Specialization information
2. User should be able to view, add, update and delete Doctor Information
3. User should be able to view, add, update and delete Patient Information

­­­

**Non-Functional Requirements**

1. The system needs to be highly available
2. The system should be with acceptable minimum latency
3. Proper relationship between doctor, patient and specialization information
4. The system should be flexible and reusable by third party applications

**Some Design Considerations**

Our System is designed with simple architecture, such that it is lighter to deploy and maintain.

1. Since, the data is maintained online, data loss to be controlled
2. Due to small scale development, the system is limited to basic CRUD operations
3. Managed to cover/include some of the standards and design patterns

**High Level System Design**

Patient Management

Doctor Management

ADMIN view, add, update, delete

Specialization Management

**Database Schema**

We need to store the registered user information in a database, which needs a relational schema.

|  |  |
| --- | --- |
| DoctorInfo | |
| PK | id: string |
|  | name: string  specializationId: string  gender: string  address: string  contact: string  joinedOn: date |

|  |  |
| --- | --- |
| PatientInfo | |
| PK | id: string |
|  | name: string  doctorId: string  age: string  gender: string  address: string  contact: string  history: string  lastVisited: date |

|  |  |
| --- | --- |
| SpecializationInfo | |
| PK | id: string |
|  | name: string |

Here in, the doctor information has dependency on respective specialization, and also patient information is dependent on respective doctor information. Due to these dependency, we are implementing Relational Database Management System. When there is no relation between data, we can go for NoSQL, which will be a straight forward approach.

**Entity Design**

Specialization ID

Doctor ID

Admin

Access records

Patient

Doctor

Treats

**Development Technology**

The core application backend is build using SpingBoot (MVC). The frontend is developed using Angular and the respective builded javascript bundle is embedded into SpringBoot in order to achieve a standard MVC application.

**Design Patterns**

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development.

Basically, Our system is designed based on MVC pattern, which is Model View Controller.

Some of the design patterns as follows:

1. **Creational Patterns**

These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case

1. **Structural Patterns**

These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities

1. **Behavioural Patterns**

These design patterns are specifically concerned with communication between objects. Changing the behavior of a function without modifying the actual code

**S.O.L.I.D Principles**

1. Single Responsibility Principle

A class should have one and only one reason to change, meaning that a class should have only one job.

1. Open-closed Principle

Objects or entities should be open for extension, but closed for modification.

1. Liskov Substitution Principle

Every subclass/derived class should be substitutable for their base/parent class.

1. Dependency Inversion Principle

Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

1. Interface Segregation Principle

A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use.