**System Design and Design Principles**

**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 build of 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:

**Domain Driven Design**

Domain-driven design (DDD) is the concept that the structure and language of software code (class names, class methods, class variables) should match the business domain. For example, since our MediBuddy processes hospital management system, it might have classes such as PatientController, DoctorInfo, SpecializationService.

Domain-driven design is predicated on the following goals:

* Placing the project's primary focus on the core domain and domain logic
* Basing complex designs on a model of the domain

**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.

**Singleton Design Pattern**

A class must ensure that only single instance should be created and single object can be used by all other classes.

To create the singleton class, we need to have static member of class, private constructor and static factory method.

* **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

In our application, Logger can be a best example to define Singleton design pattern.

**private** **static** **final** Logger ***LOG*** = LoggerFactory.*getLogger*(DoctorController.**class**);

**Prototype Design Pattern**

In our system, the respective pattern is implemented as mentioned below,

In Specialization Module, a factory class “SpecializationService” of type interface is created. It contains only declaration and the actual implementation are performed in the child class “SpecializationServiceImplementation” class.

**public** **class** SpecializationServiceImplementation **implements** SpecializationService {...

**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.

**Bridge Design Pattern**

A Bridge Pattern says that just "decouple the functional abstraction from the implementation so that the two can vary independently".

**Usage of Bridge Pattern**

* When you don't want a permanent binding between the functional abstraction and its implementation.
* When both the functional abstraction and its implementation need to extended using sub-classes.
* It is mostly used in those places where changes are made in the implementation does not affect the clients.

Interface file:

**public** **interface** DoctorService {

List<DoctorInfo> getDoctorInfo();

}

Implementation file (Bridge file):

**public** **class** DoctorServiceImplementation implements DoctorService {

@Override

**public** List<DoctorInfo> getDoctorInfo() {

**if**(doctorDao.count() > 0) {

**return** (List<DoctorInfo>) doctorDao.findAll();

}**else** {

**throw** **new** DataNotFoundExceptionResolver("No Data Found");

}

}

}

Controller class that uses Interface file:

**public** **class** DoctorController {

@RequestMapping(value = "/getDoctorInfo", method = RequestMethod.***GET***)

**private** List<DoctorInfo> getDoctorInfo(){

***LOG***.info("Getting Doctor Info!!!");

**return** doctorService.getDoctorInfo();

}

**}**

**Decorator Design Pattern**

The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

**Usage of Decorator Pattern**

* When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
* When you want to add responsibilities to an object that you may want to change in future.
* Extending functionality by sub-classing is no longer practical.

In our system, “PatientService” interface has its own method declarations, whereas the “PatientServiceImplementation” class implements PatientService interface and overrides the methods on runtime.

**public** **interface** PatientService {

List<PatientInfo> deletePatientInfo(String id);

}

**public** **class** PatientServiceImplementation implements PatientService {

@Override

**public** List<PatientInfo> deletePatientInfo(String id){

**...**

**}**

}

**Behavioural Patterns**

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

In our system, we have extended the “RuntimeException” class and created our own modified resolver class, where we have reused the parent class functionality based on our requirement without modifying the existing code base.

**public** **class** DataIntegrityExceptionResolver **extends** RuntimeException {

**public** DataIntegrityExceptionResolver(String exception) {

**super**(exception);

}

}

**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.

In our system, we used "CommonExceptionResolver” class which act as a single class that serves all over the application for handling various exceptions

**public** **class** CustomExceptionResolver {

@ExceptionHandler({ DataIntegrityExceptionResolver.**class** })

**protected** ResponseEntity<Object> customKeyConstraintExceptionResolver(DataIntegrityExceptionResolver ex) {

ResponseInfo error = **new** ResponseInfo();

error.setStatusCode(547);

error.setStatus(**false**);

error.setMessage(ex.getMessage());

***LOG***.error("{} {}",error.getStatusCode(),error.getMessage());

**return** **new** ResponseEntity<>(error, HttpStatus.***INTERNAL\_SERVER\_ERROR***);

}

}

1. **Open-closed Principle**

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

In our system, there are some cases, where we need to handle runtime errors and exceptions with our own userdefined error messages. In such case, the predefined “RuntimeException” class in extended and we implement our own custom “ExceptionResolver” to handle the exceptions with userdefined messages. Here, the class “ RuntimeException” is open for extension, but closed for modification.

**public** **class** DataNotFoundExceptionResolver **extends** RuntimeException {

**public** DataNotFoundExceptionResolver(String exception) {

**super**(exception);

}

}

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.

In our system, even though Specialization and Doctor modules has relation on each other, still “SpecializationController” and “DoctorController” does not depend on each other, instead they are dependent on “SpecialziationService” and “DoctorService” interfaces respectively

**public** **class** SpecializationController {

@Autowired

SpecializationService specializationService;

}

**public** **class** DoctorController {

@Autowired

DoctorService doctorService;

}

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.

In our system, all CRUD operations such as view, add, update and delete is common to both Specialization, Doctor and Patient modules. Anyhow we are not using same interface for all the modules. Instead, we are writing individual interface for each modules separately.

**Specialization Module:**

**public** **interface** SpecializationService {

...

}

**public** **interface** SpecializationServiceImplementation implements SpecializationService {

...

}

**Doctor Module:**

**public** **interface** DoctorService {

...

}

**public** **interface** DoctorServiceImplementation implements DoctorService {

...

}

**Patient Module:**

**public** **interface** PatientService {

...

}

**public** **interface** PatientServiceImplementation implements PatientService {

...

}

**Key Features implemented in the application**

* Logger
* Junit Test cases
* MVC pattern, and certain design patterns, design principles
* User interface to perform CRUD operations
* SpringBoot services that can be utilized by any third party applications

**Future Enhancements**

Our MediBuddy application is designed in such a way, that any future modules or enhancements can be added in a structural manner. Since, the application is based on domain driven design, it is easily understandable and helps any future developers to enhance the application based on new requirements.

**Conclusion**

MediBuddy is a Hospital Management System, that provides basic CRUD operations/services over Doctor, Specialization and Patient records. The application is light weight to handle, execute and deploy, since it is designed over certain proper design patterns and principles.