1. Present and explain how your project satisfies the SOLID design principle.

SOLID principles are the design principles that enable us manage most of the software design problems.

SOLID Acronym:

- S: Single Responsibility Principle (SRP)
- O: Open closed Principle (OSP)
- L: Liskov substituition Principle (LSP)
- I: Interface Segregation Principle (ISP)
- D: Dependency Inversion Principle (DIP)

1.1, Single Responsibility Principle

" A class should have only one reason to change". Every module or class should have responsibility over a single part of the functionality provided by the software and that responsibility should be entirely encapsulated by the class.

E.g: Our project use JpaRepository class to:

+ Open database connection
+ Fetch data from database
+ Adopt ORM to manage queries on database

1.2, Liskov Substitution Principle

"Objects in a program should be replaceable with instances of their sub-types without altering the correctness of that program". If a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module. We can also state that Derived types must be substitutable for their base types.

E.g: Let's consider an ResponseEntityExceptionHandler parent class in our project:

```
public abstract class ResponseEntityExceptionHandler {

    protected ResponseEntity<Object> handleMethodArgumentNotValid(
MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status,
WebRequest request) {

        return handleExceptionInternal(ex, null, headers, status, request);
    }
}
```

Now let's consider the CustomerRestExceptionHandler class which extend ResponseEntityExceptionHandler:

```
public class CustomRestExceptionHandler extends ResponseEntityExceptionHandler {
    @Override
    protected ResponseEntity<Object>
handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders
headers, HttpStatus status, WebRequest request) {
```

```
        String message = "";
        for (FieldError error : ex.getBindingResult().getFieldErrors()) {
                message = message + error.getField() + " " + error.getDefaultMessage() + "
and ";
        }
        for (ObjectError error : ex.getBindingResult().getGlobalErrors()) {
                message = message + error.getObjectName() + " " +
error.getDefaultMessage() + " and ";
        }
        message = message.substring(0, message.length() - 5);
        return new ResponseEntity<>(new ApiResponse(false, "argument_not_valid",
message), HttpStatus.OK);
}
```

Now, wherever in our code we were using ResponseEntityExceptionHandler class object we must be able to replace it with CustomRestExceptionHandler without exploding our code. What do we mean here is the child class should not implement code such that if it is replaced by the parent class then the application will stop running.

1.3, Open/Closed Principle

“Software entities should be open for extension, but closed for modification”. The design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

Let’s look at an example in our code, the class DateAudit is written for creating all other model with two characteristic createdAt and updatedAt by extending class DateAudit without modifying it. Developers can easily extend this class and create their own custom model.

```
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
@JsonIgnoreProperties(
value = {"createdAt", "updatedAt"},
allowGetters = true
)
public abstract class DateAudit implements Serializable{

private static final long serialVersionUID = 1L;

        @CreatedDate
        @Column(nullable = false, updatable = false)
        public Instant createdAt;

        @LastModifiedDate
        @Column(nullable = false)
        private Instant updatedAt;
```

```
        public Instant getCreatedAt(){
                return createdAt;
        }

        public void setCreatedAt(Instant createdAt){
                this.createdAt = createdAt;
        }

        public Instant getUpdatedAt(){
                return updatedAt;
        }

        public void setUpdatedAt(Instant updatedAt){
                this.updatedAt = updatedAt;
        }
}

@Entity
@Table(name = "users", uniqueConstraints = {
@UniqueConstraint(columnNames = {
"username"
}),
@UniqueConstraint(columnNames = {
"email"
})
})
public class User extends DateAudit{
        ...
}
```

1.4, Interface Segregation Principle

“Many client-specific interfaces are better than one general-purpose interface”. We should not enforce clients to implement interfaces that they don't use. Instead of creating one big interface we can break down it to smaller interfaces.

For example, in our communicating between client and server, we build a RESTful API WebService that provide many API based on functionality and requirement.

1.5, Dependency Inversion Principle

One should “depend upon abstractions, [not] concretions" . Abstractions should not depend on the details whereas the details should depend on abstractions. High-level modules should not depend on low level modules.

For doing this purpose, we apply the passive MVC pattern and 3-tire layer structure for our project.

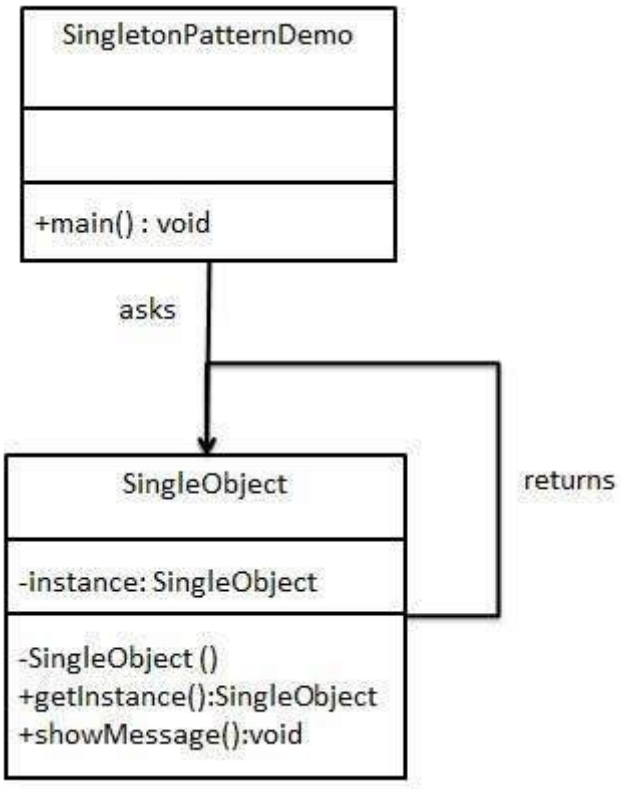
2. Which design patterns do you use for your project ? Why and how ?

2.1, Singleton

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

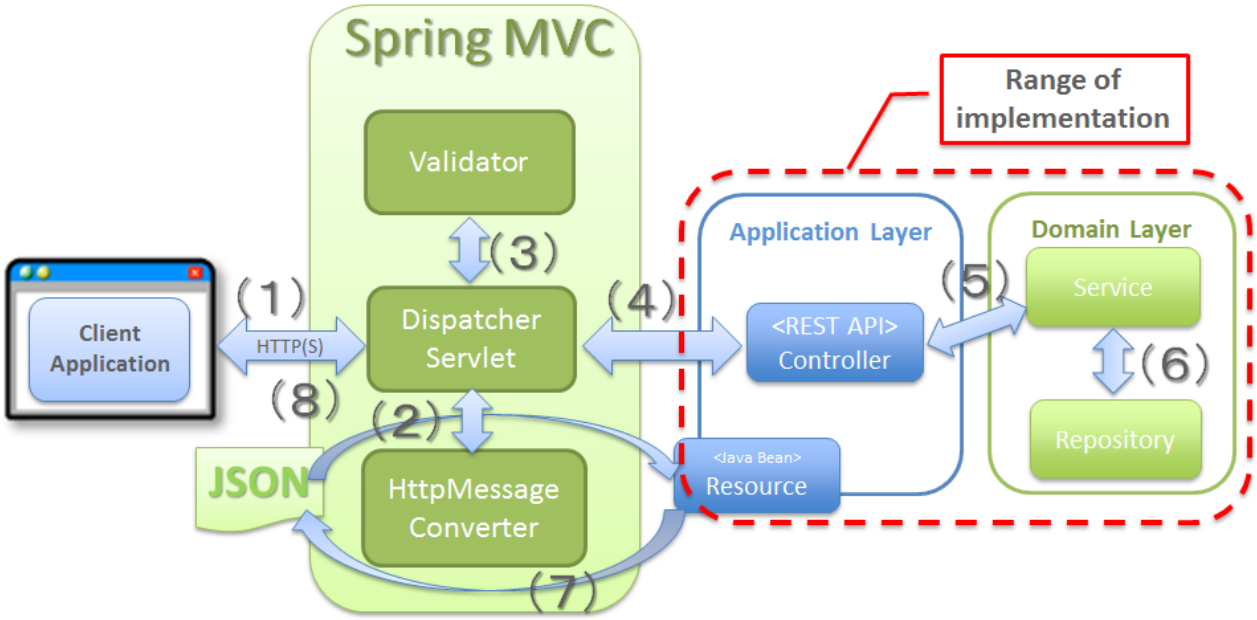
This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

SingleObject class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.



2.2, MVC

For server-side, we apply Spring MVC with RESTful Web Service structure



And also apply MVC for cliend side: