Website = Static (Readable)
Web App = Dynamic (Read/write both)

Library = Pre written code we can use in an application

framework = minimum codebase /standard

Why Jalla? (because it is a tough competitor) / (Jalla is always Up to date)

Cricjaya by hand, all shapes will have different

Cricjaya by Cricjaya maker

H2 -> My SQL DataBase

Barkers

Controller,
Service,
Data Access layer,

Hotel

Reception
Waiter
Chef

Why Spring ? b

DI = Dependency Injection

IOC = Inversion of Control

Node .JS -> is not good for CPU intensive Task

(marriage) Tight Coupling -> Class are Dependent

(Relationship) Loose Coupling -> Not are Dependent

Spring bring loose coupling automatically

Spring is a framework

Class A
{
B b;
tight coupling
}

Class B
{
}

Design Principal

DI => Inject the classes / object where it needed with out tight coupling

IOC (=) ↑ Spring control the application (DI) for new

Design Pattern

help us to Create loosly couple application

Done By Spring

-> Create Class B
-> Inject Class B in A
-> Create Class A

Maven ← Build Tool
Dependency management

a collection of Java dependencies

Project management
Build Tool

manage the structure the project
we can generate JAR/WAR file using this

JAR → compressed version of all Java class in Java Application

copy dependency in local repository so we can use it as much we can

Maven's POM.xml ← Provide instruction to download Dependency

Spring

⇒ Spring Core
⇒ Spring - context
⇒ apache - common - logger

Coupling

↓
Problem of tight coupling (samadhan)
① code interface
② factory Design Pattern
} Extra things

Giving instruction to spring
① Spring.xml
② Programatically using JAVA (flexible)
③ new annotations (@) ← give extra information about the class
↑ a meta data

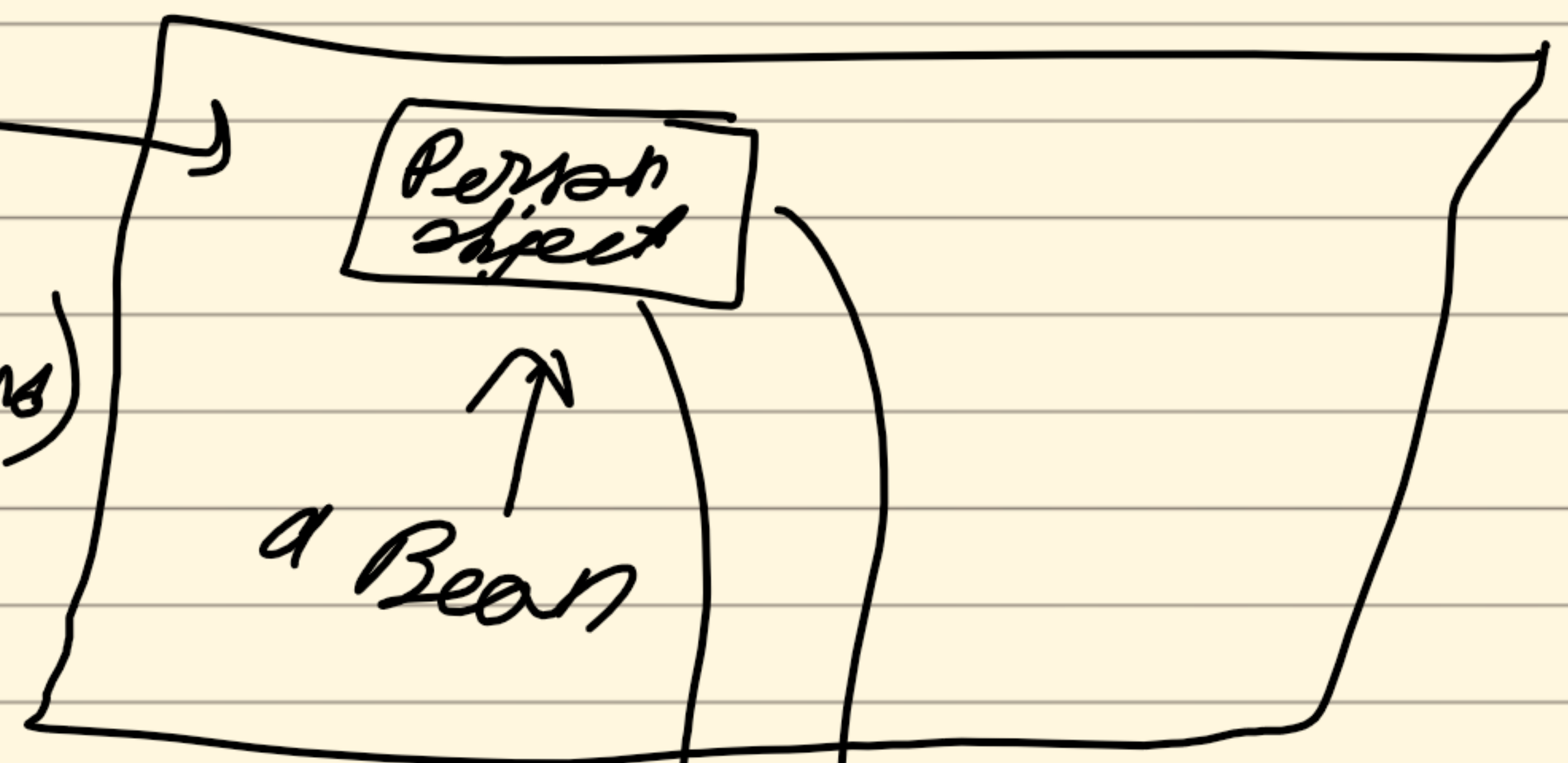
@Component ← → tell spring to create object and keep ready
→ only in class

@Component ("myPerson") ← will change name
class Person{

≡

}

Spring Container / IOC Container



(Cruord) → Application Context, helps in fetch beans

→ ID (same as class name with first letter small)

→ to (Person)

an Interface

ApplicationContext ctx = new AnnotationConfigApplicationContext("");
com.Application ← Base Package name

Person person = (Person) ctx.getBean ("person")

→ return Object class & we have to typeCast

↑ Root Path of folder

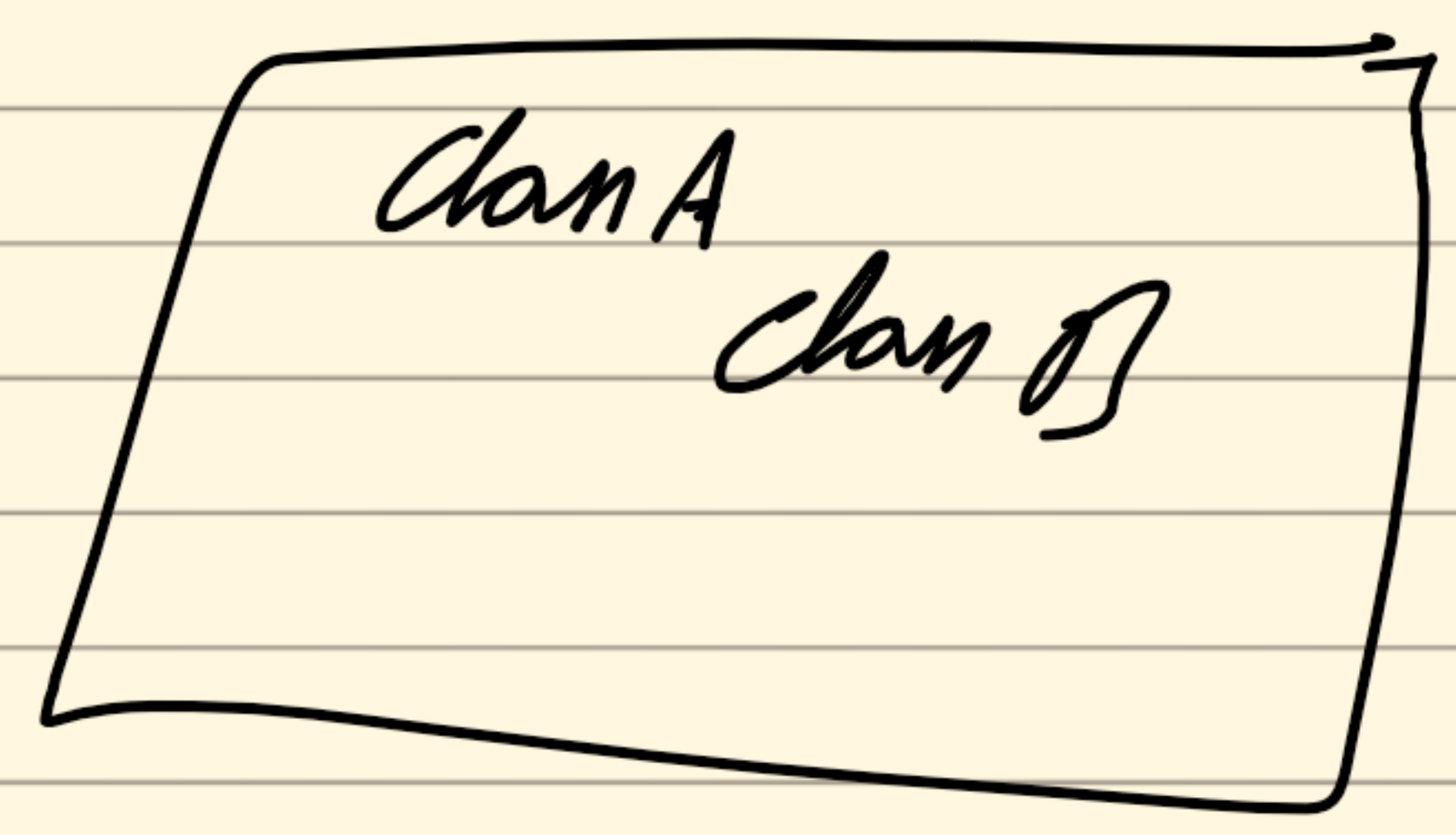
By default Beans are Singleton design Pattern

@Autowire ← inject the object

@ Qualifiers ("classA") ← give hint to spring to chose which Bean (it does not change name)

no unique bean exception

Spring Container



interface = class A()

interface
class A   class B

Scope of Bean

1. singleton ← Default (one instance only)
2. Prototype ← (new instance on each call)
3. Request ← (new Bean for each request)
3. Session ← session Base (new session new Bean)
4. Global session ← Global HTTP session Base

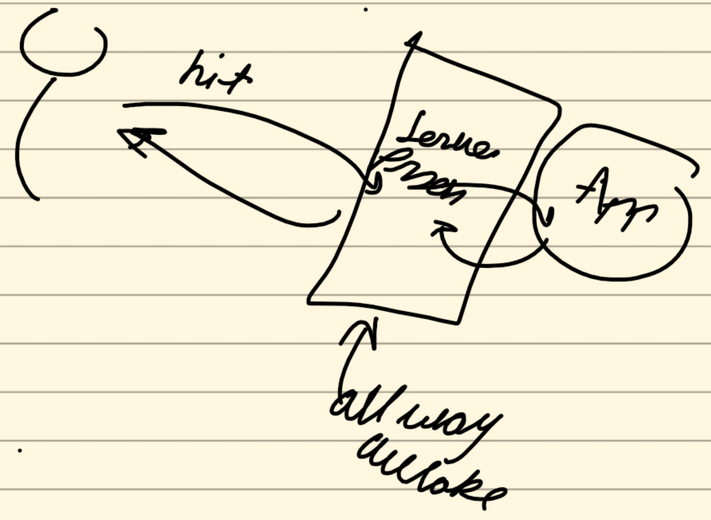
@Scope ("prototype")

Spring-Boot → Gesture is enough

⇒ auto configuration

Spring Problem / too many configuration

Server = server staring process



ways to install

1) Spring-Bot CLI
2) maven
3) IDE
4) Spring initializer

web server
Internet serving traffic

Application Server
Non-Internet Serving traffic

Eg: Tomcat / Jetty
for Jolla base Application

JAR → Zipped version of Jars files
WAR → for deploying Application in Server

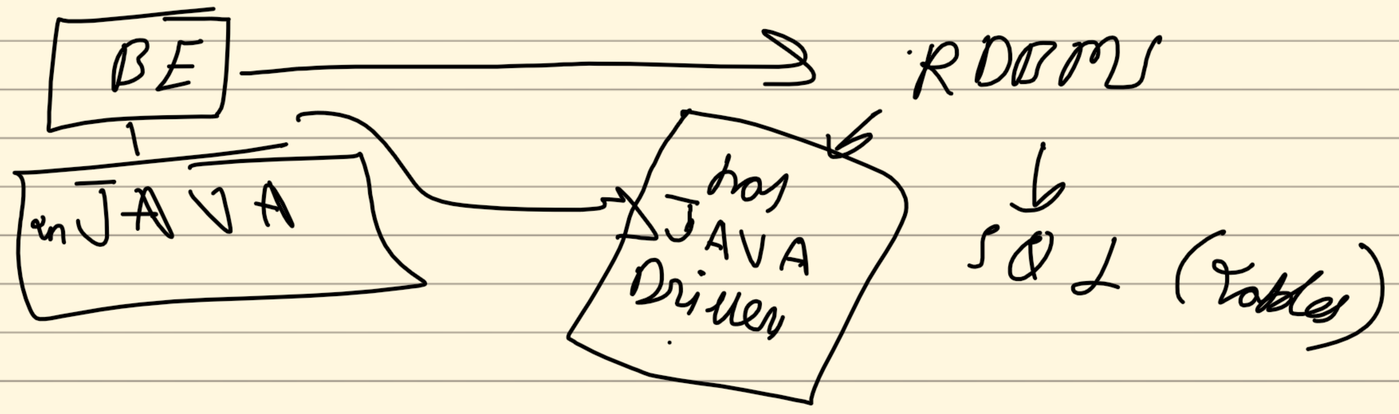
Stand alone app embedded Server
for fast dev

every Spring Boot Project have Parent Spring-Boot-Starter-Parent ↘ in Pom.xml

@ SpringBootApplication
- @ SpringBootConfiguration (do auto configuration)
- @ EnableAutoConfiguration
- @ ComponentScan ← scan the packages and subfolder for making bean

P.T.O

DATA Access layer (DAL)



" JDBC " API → Application Programming interface (contract/Rule to follow in any kind of Relationship)

JAVA DATABASE Connectivity

JDBC API | JDBC Drivers
(Rules) (actual implementation)

↓ list of code/multiple steps for small application

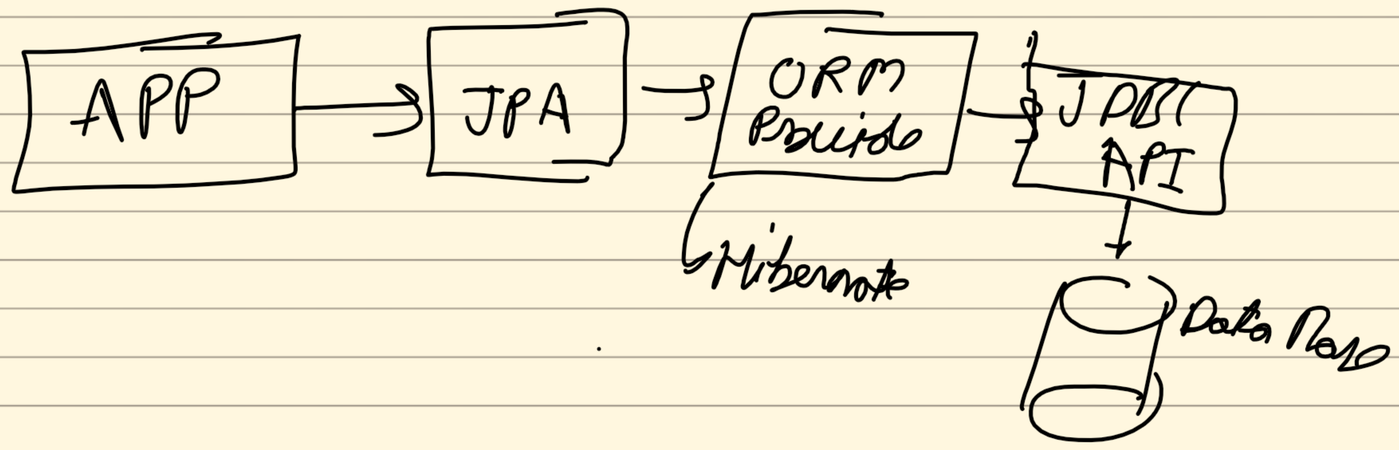
Hibernate company → Introduce

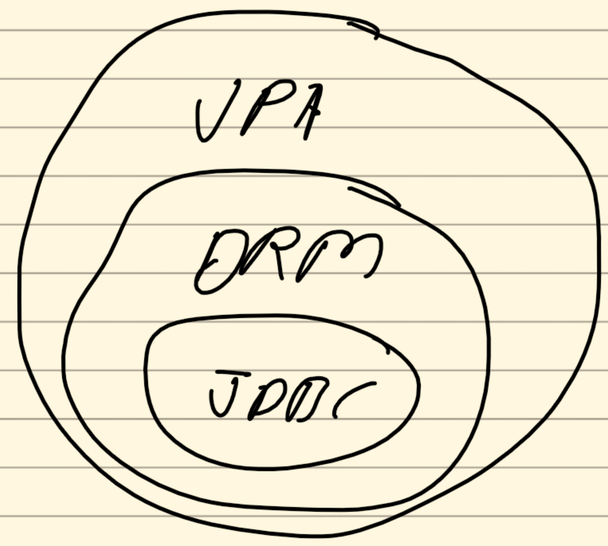
ORM (Object Relation Mapping) ← Act like a translator, all commands in Java

↳ internally Use JDBC

JPA (Java Persistent APIs)

↳ a standard set for ORM





@Entity ← create table

class Data{ @ID Primarykey,

Data,
Data,
Data

}

only using these tabs we can Create Table

movieName → MOVIE_NAME

@Table("Movie_Table")

↘ Create table name

@Column(name="custom name", length=500, nullable=false)

Constraint →

@GeneratedValue(strategy=GenerationType.AUTO)

← default (Framework decide which to choose)

. AUTO
. IDENTITY ← Increment (mySql, MSSql)
. SEQUENCE ← use database Sequence (Oracle
. TABLE ← Create a new Table and increament it (all Database

Spring_Data ← comes with Spring Boot

Data Access object

interface MovieDAO extent JPARepository<Movie, Integer>

class Id

automatically give all Methods to do CURD operation

MovieDao.save(movie);

MovieDao.findByID(ID).get()

this will get object from Database

Optional<>

this use to Bypass Null Point exception

Simple query

in Movie dao

findByMovieName(name) ← in interface and Spring will implement this

Relation ship

→ one - to - one
many to one
one to many
many to many

@ Many to one — Theatre → City

@ ManyToOne
private City city ; → City - City - ID ← in theatre table

@ JoinColumn (name = "City") → with change "City - City - ID" → "City"
, nullable = false

in theatre — in City

used as @Column for Relation

(now making bidirectional relation
& city also know about theatre

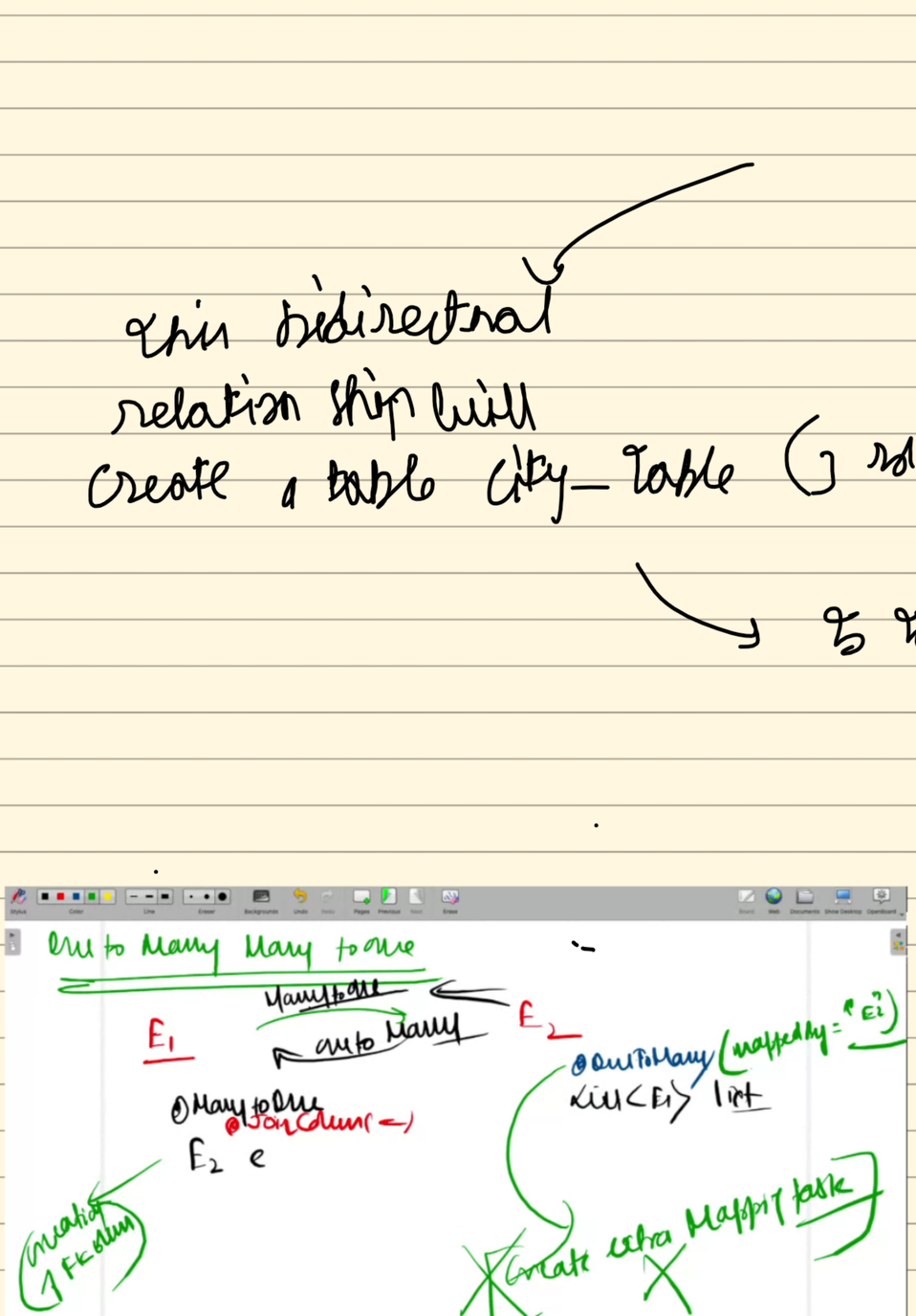
Theatre ⟷ City

@ OneToMany
private Set < Theatre > theatres ;

this bidirectional
relation ship will
create a table city_Theatre (3rd table)

→ to Table this we use (mappedBy)

@ OneToMany (mappedBy = "City")



Fetching Strategies (eager, lazy)

one to one
many to one
} eager ← default

many to many
one to many
} lazy

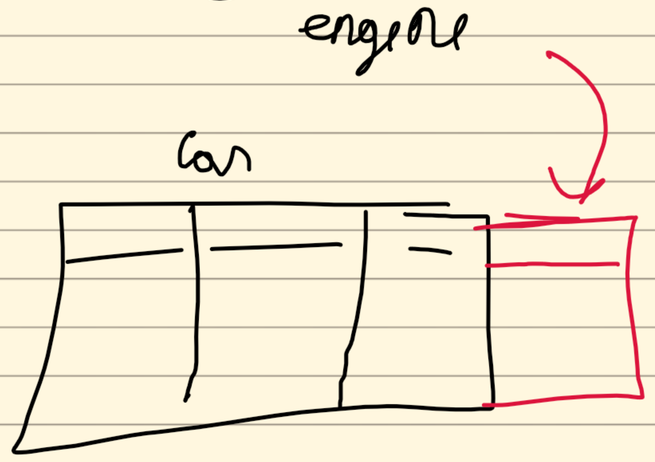
load data on demand
(lazy loading)

@OneToMany (fetch = FetchType.EAGER)

one to one

Car ⟶ engine

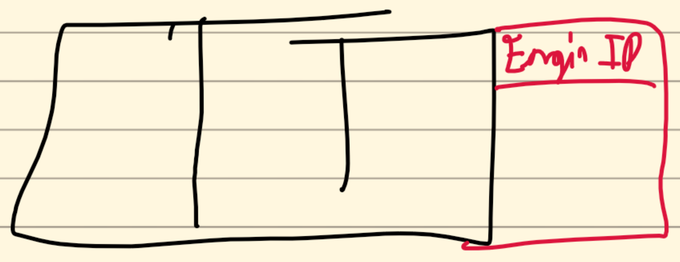
@JoinColumn("engine")
@OneToOne.



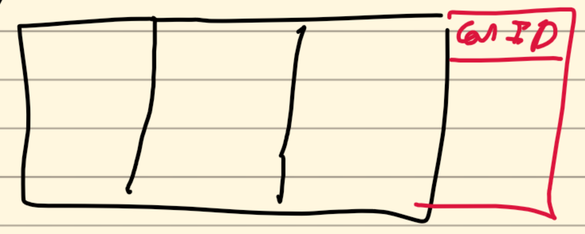
unidirectional

Car → engine

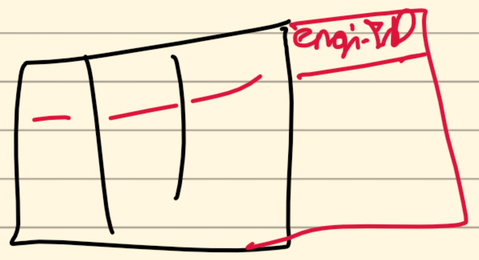
@OneToOne
Car

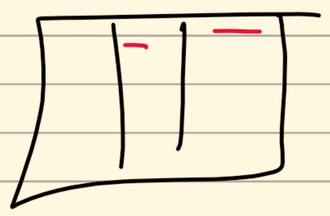


engine



@OneToOne("engine")
Car



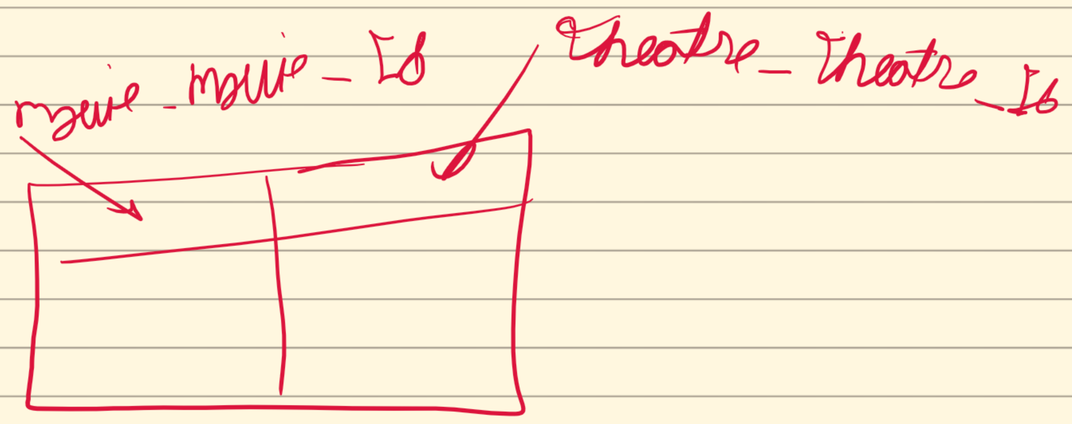


"bidirectional at the level of object" we want but not in database

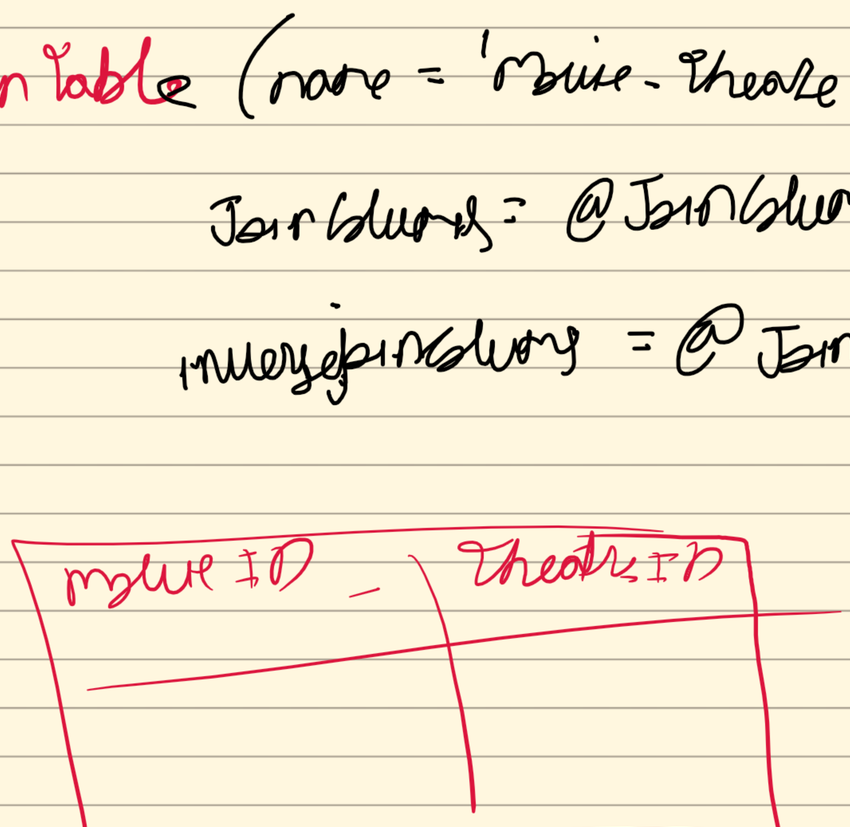
many to many

movie → Theatre

@ManyToMany
List<Theatre> theatres

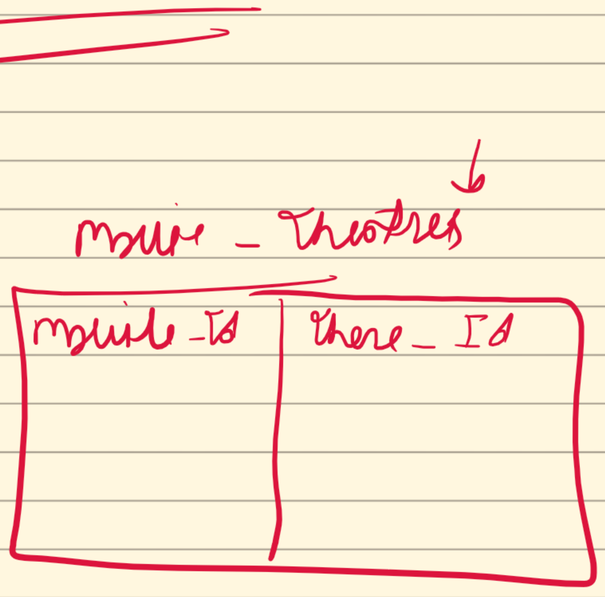


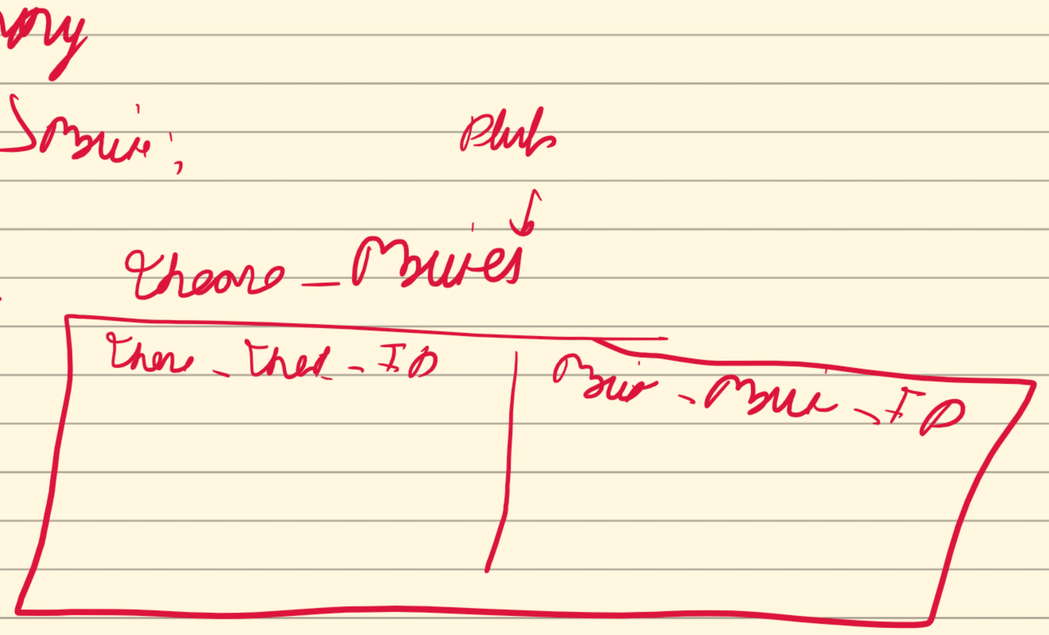
@JoinTable (name = "movie_theatre",
joinColumns = @JoinColumn(name: "movie_ID"),
inverseJoinColumns = @JoinColumn(name="theatre_ID")



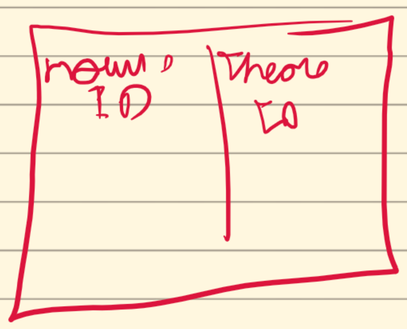
Bidirectional

@ManyToMany
List<Movie> movies;





(@ManyToMany (mapped by = "theatres")



make table for attribute

@ CollectionTable (name = "author_contact_number" , joinColumns = @JoinColumn (name = "author_ID"))
@ ElementCollection
Private Set <Integer> phoneNumber ;
@ Column (mobile_number

Cascading (to delet Parent you also have to delete all child)

Author → Books

C1 → B1, B2, B3

delete C1 with attach

B1, B2, B3, C1 ← this order to delete C1

Referential Integrity

JPA Cascading

Type of Cascading

operation perform on parent also child

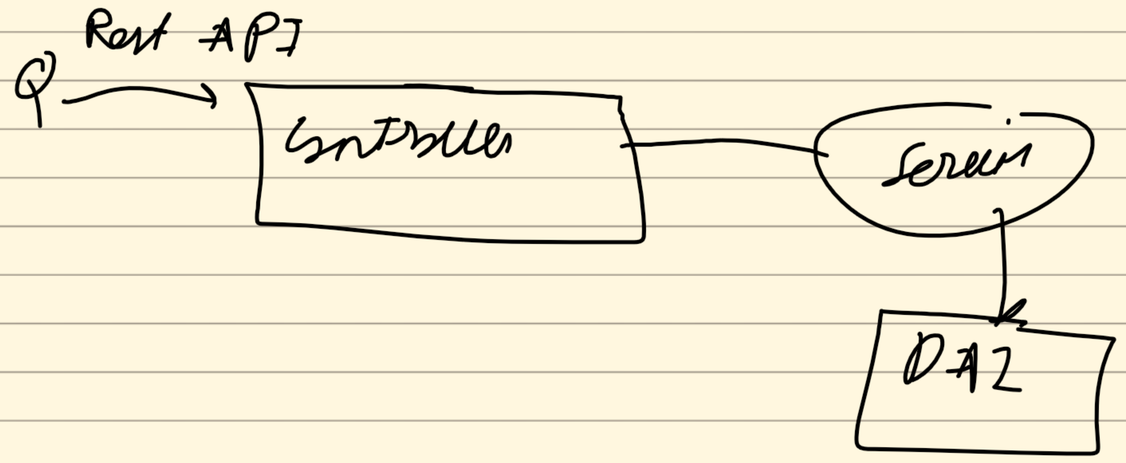
Parent → child

1) Persist ←

2) Remove → if remove Parent, child also remove
3) Refresh → if loading Parent child also loaded

4) merge
5) Detach } entity state

6) All what is happening to Parent will also happen to child



entities.

model → model Dao → DAL/DAO

user → User Dao → DAL/DAO

city → City Dao → DAL/DAO

Service

model service (I) interface → Implementation

↳ model service Impl () class

use service (I)

↳ use service Impl ()

City service (I)

↳ city service Impl

Create Bean

@Service ← say that this is a Service

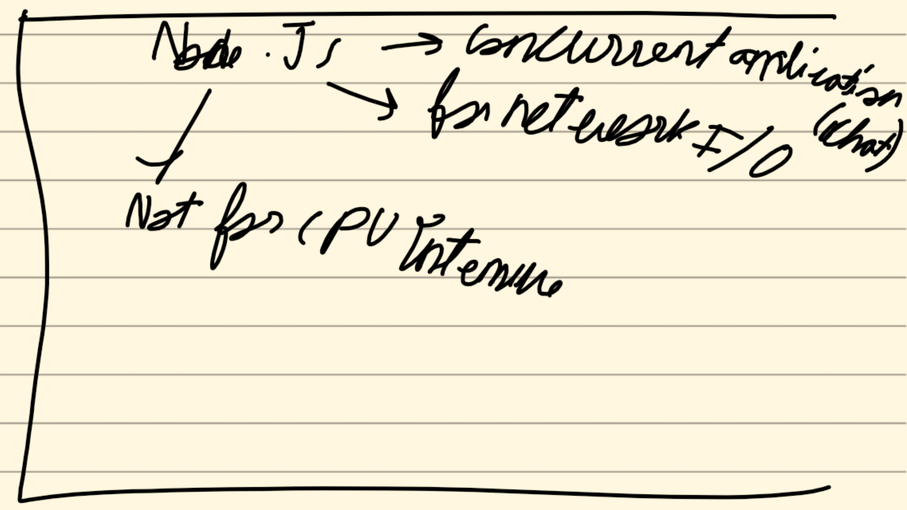
class model service Impl implements model service {

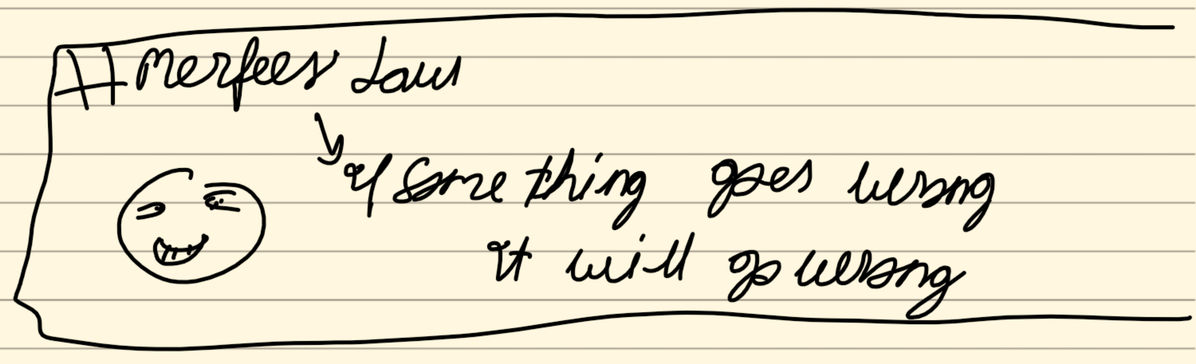
@Autowire

model Dao model Dao ;

}

we can use this Dao





Testing

@Service
Interface InitService   all entities
{ = Dao;
@Autowire
Public void init();

}

Main () {

}

@Bean
CommandLineRunner init(InitService initSer) {
return args -> {
initService.init()
};
}

What you right will run first by Spring boot

Sequence
-> Jar created
Jar placed inside Tomcat   Jar
Tomcat start -> a new Java process   Tomcat
H2 database start.
Init runs

Desire vs Expected result difference

=> Testing = Identifying Problem is (Testing)

=> Debugging = Identifying Root cause of the Problem

(1) Requirement
↳ (2) scope / Planning
↳ (3) Design / Architecture
↳ (4) implemented / code
↳ (5) Tester (tested)

Problems : we can release our targets (because tester involve very late)

solution: Testing code before implementation by test cases

TDD (run implement test cases using TDD) -> TDD
↳ functional Design document
↳ Technical ~~Details~~ Design Document

JUnit framework ← to test Java app

class TestDemo {

@Test ← a test method
void myFirstTest() {

}

@BeforeEach ← execute before each test
void beforeEach() {
System.out.println("BeforeEach");
}

@AfterEach
void afterEach() {

}

@BeforeAll
void executeBeginning() { ← before all test

}

@AfterAll
void executeEnding() {

}

from JUnit framework

Assertions.assertEquals(actualResult, expectedResult);
Param ↘ ==
fail X !=

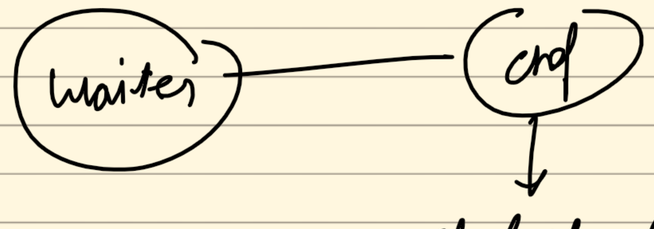
AssertionsassertNotNull(==
this should not be null

@SpringBootTest
↓
this will be defined in test folder as
@SpringBootApplication

Unit Test = Testing Double only one class

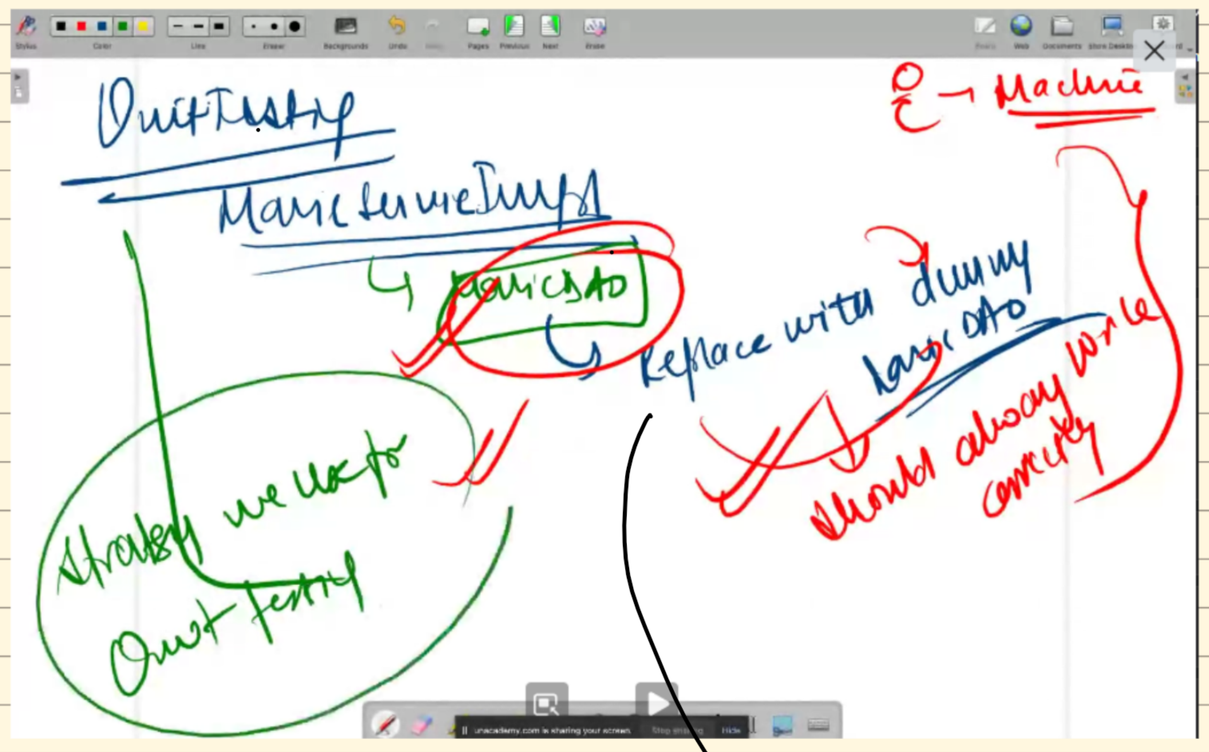
Integration Testing = Testing Integration

How to test waiter



chef should not perform bad

So that we can test waiter



Replace with dummy so that it always work well

Mockito ← replace actual dependency with dummy

a library ← mock dummy dependency return correct result

@Mock

Private MovieDao movieDao

↓

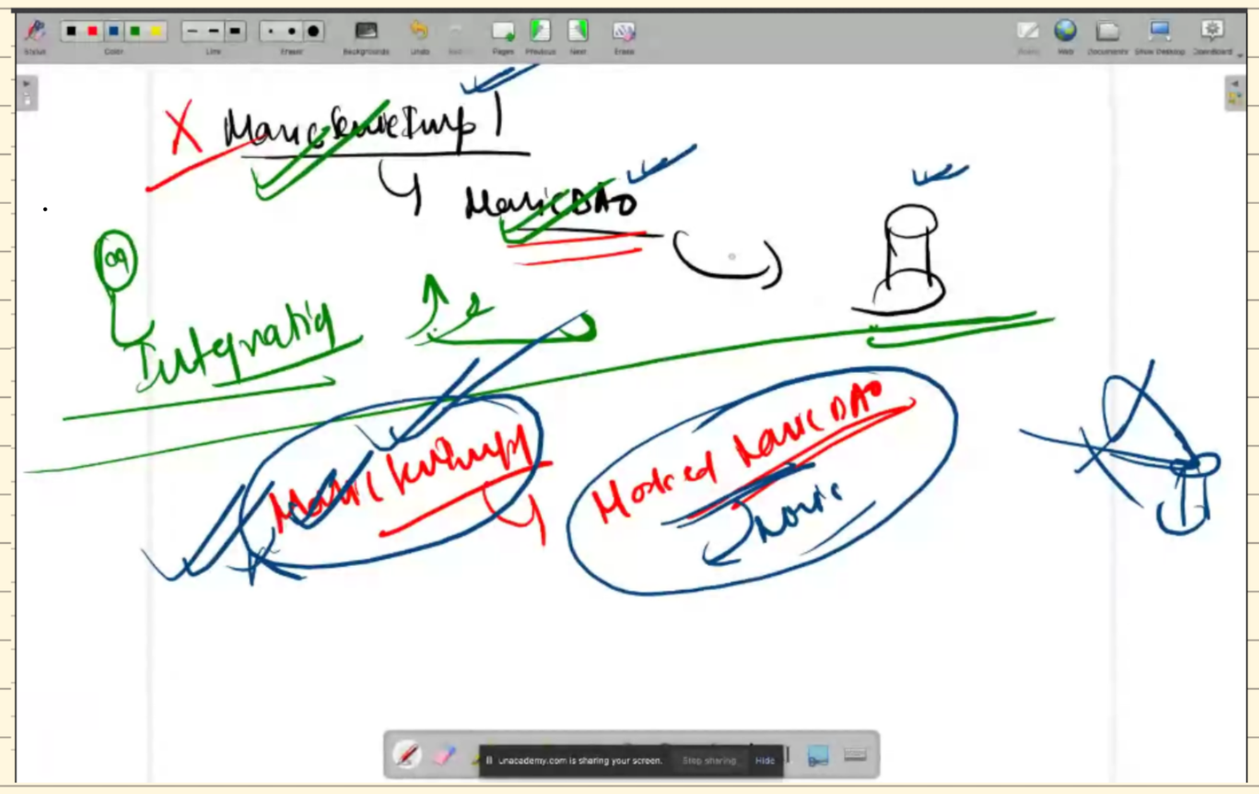
@InjectMocks

Private MovieServiceImpl movieService

movie = new Movie();

Mockito.when(movieDao.save(movie)).thenReturn(movie)

↑ to add functionality




```
@SpringBootTest
Public class CityControllerTest

@Test
Public void testGetAllCities()

}

@Mock
private CityRepo cityRepo;

@InjectMocks
Private CityService cityService;

@BeforeEach
Void setup()
{
MockitoAnnotations.initMocks(this);
city = new City("Bangalore");
when(cityRepo.findAll()).thenReturn(city);
}
```

multiple dependencies

```
@Mock
class A B;
@Mock
class C;
@InjectMock
private class A;
```

class A
↔ class C

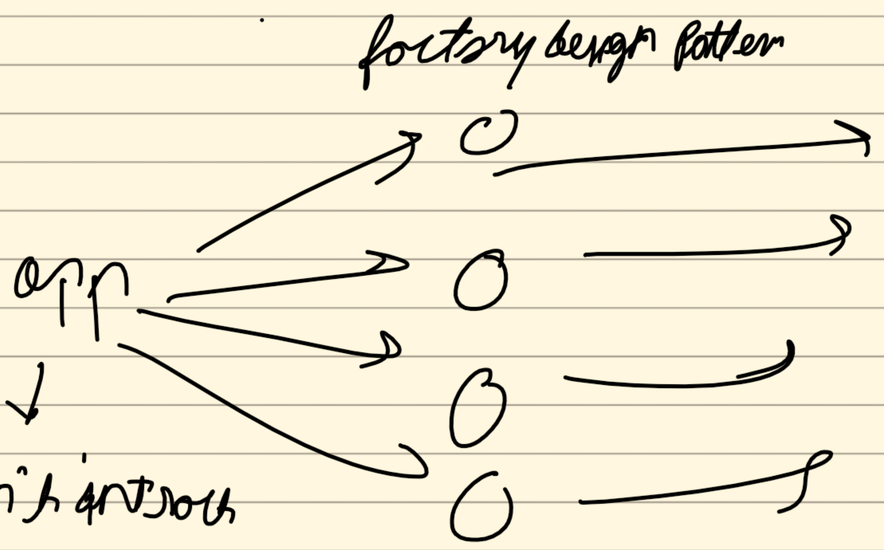
Logging (Keep record of all the events that happens during execution of program)

Logging levels

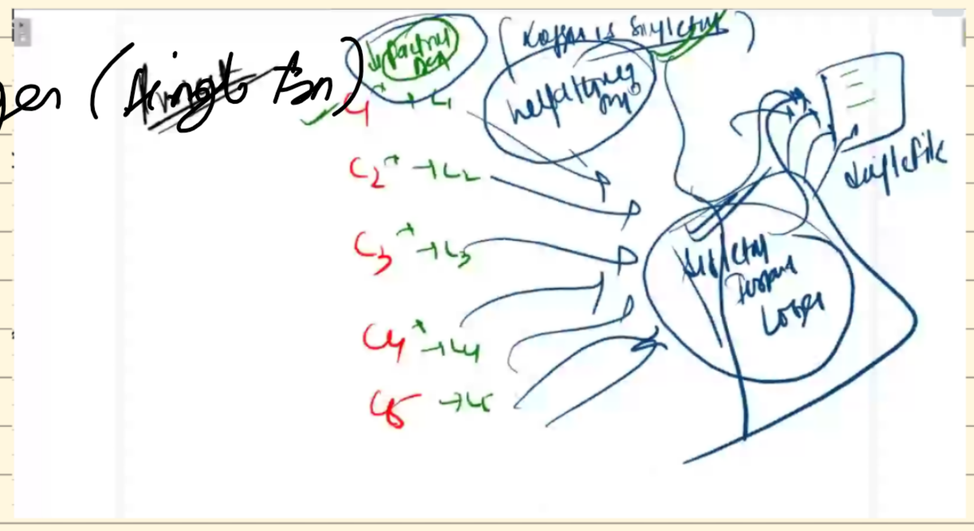
Root logger
logging.level.root = DEBUG
(property in file)

1. Debug → log everything (make log heavy)
2. info → important messages
3. warn → warnings
4. error → error but not bring system down
5. fatal → can bring system down

factory design pattern



Logger (Slf4j)



MovieBookingApp

{

Each class have one logger

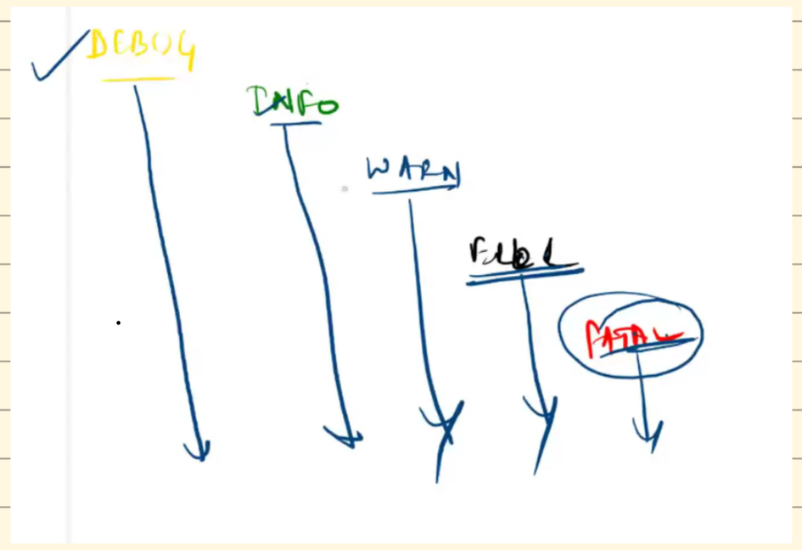
```
private static final Logger LOGGER = LoggerFactory.getLogger(MovieBookingApplication.class)
```

}

#

LOGGER.debug("writing for")

LOGGER.info("write in info")



all below will also log

logging.file.path=app.logs

all log in one file

Centralise log

1 log at one place

REST → why we need

because → SOA → Service oriented Architecture

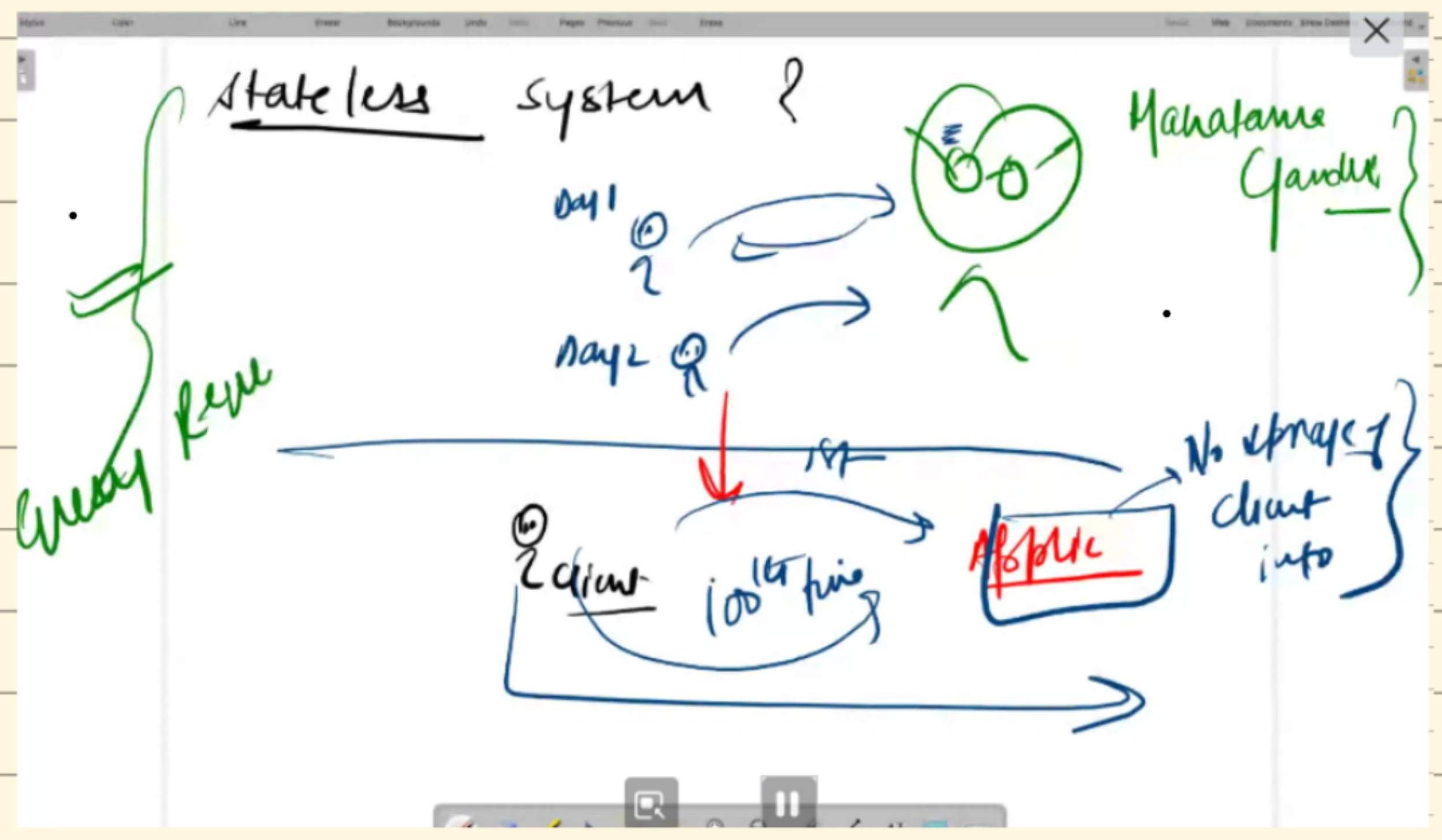
Service requester ⇄ Service Provider

Set of service requesters

Set of Service Providers

REST

- Stateless (every request is new request)



Communicate (exchange data) → happe in form of Byte Array / Serialization

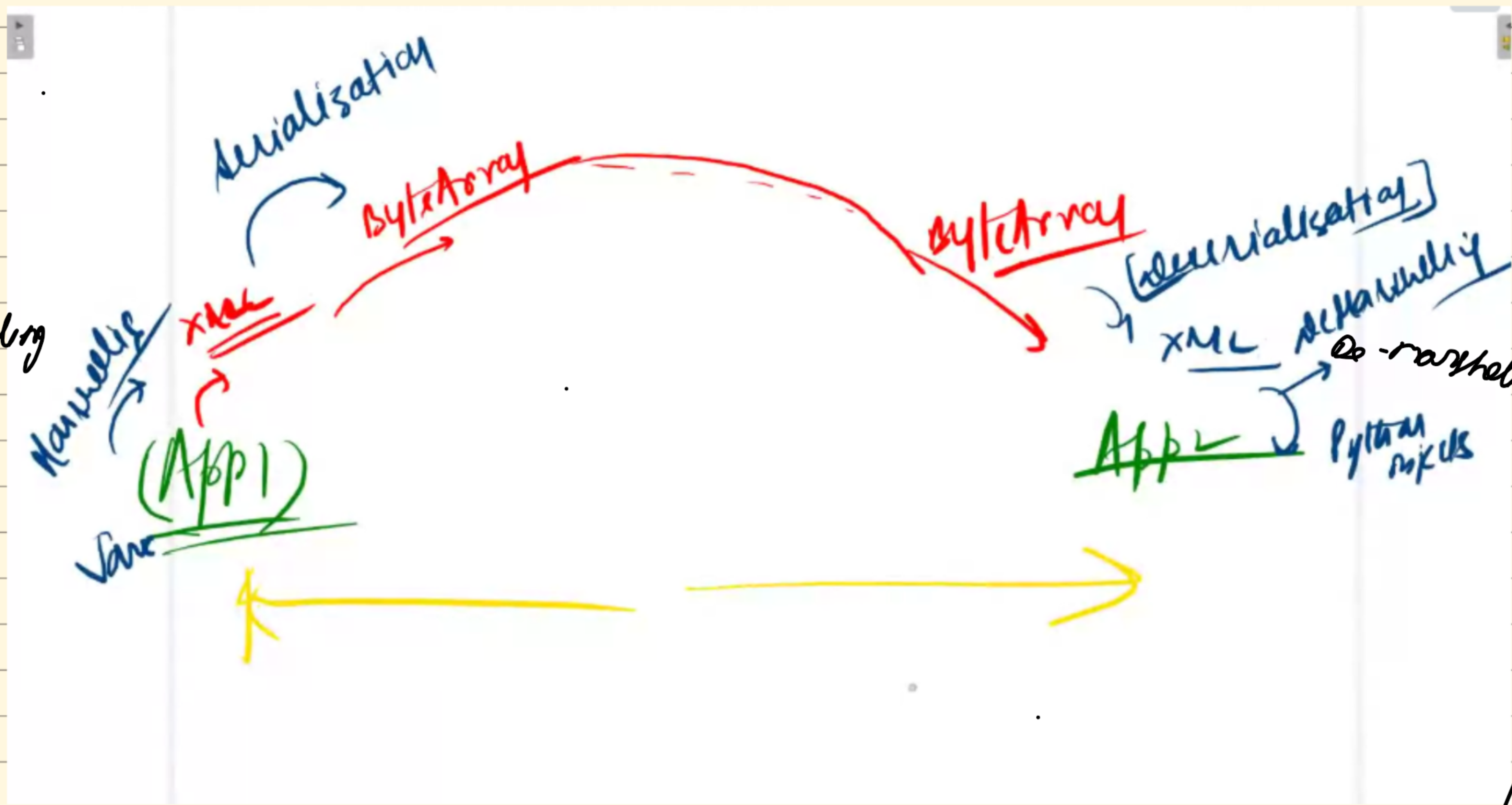
depen.
Use (XML) → Soap (Simple object access Protocol) → Before Rest ★ slow because of meta data

this is not JS ← JS because the person who develop
Depens (JSON)

(Rest)
Representational State transfer

XML/JSON
language Neutral

gRPC end Points
→ gRPC (By google)

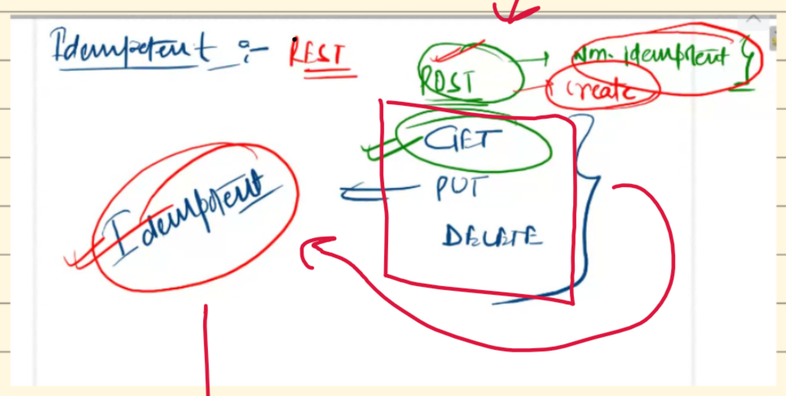


Marsheling

De-marsheling

No use @ over "interfaces" because spring will not able to make beans because interface cannot make instance of interface

Idempotent



Create a Resource

No side effect

@RestController ——— {@Component

Root of base Path

```java
@RestController
/**
 * 1. @Component
 * 2. Give hint to Spring, that whenever a REST
 * comes, make sure this class is informed
 *
 */
public class MovieController {
```

[Server. servlet. context-Path = /mb/vi]

@ RestController
@ RequestMapping ("/movies")
class MovieController {

}

Get /mb/v1/ movie/greet

@GetMapping ("/greeting")
Public ResponseEntity ubrb () {

return new ResponseEntity (body: , HttpStatus:)
}

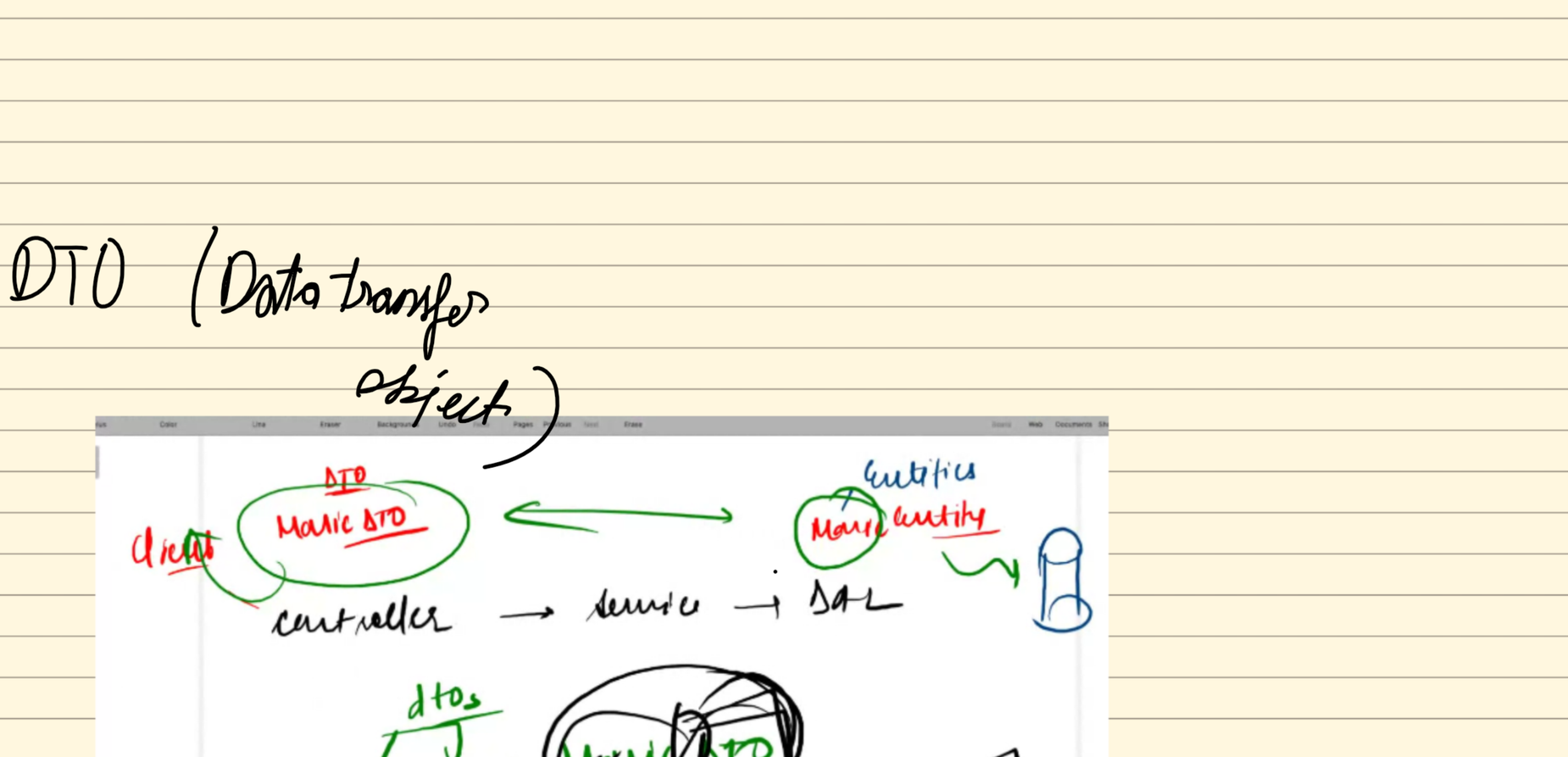
@PathVariable

```
 */
@GetMapping("/{movieId}")
public ResponseEntity getMovieBasedOnId(
    @PathVariable(name="movieId") int movieId){

}
```

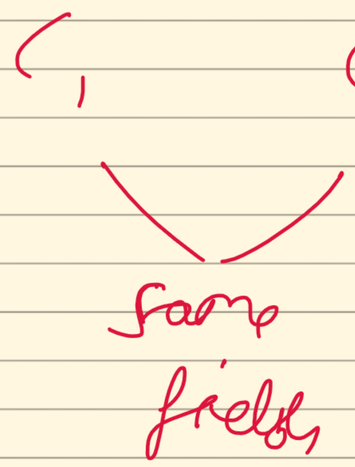
```
 */
@GetMapping("/{movieId}")
public ResponseEntity getMovieBasedOnId(
    @PathVariable(name="movieId") int movieId) throws MovieDet
    Movie movie = movieService.getMovieDetails(movieId);
    new ResponseEntity(movie, HttpStatus.OK);
}
```

#

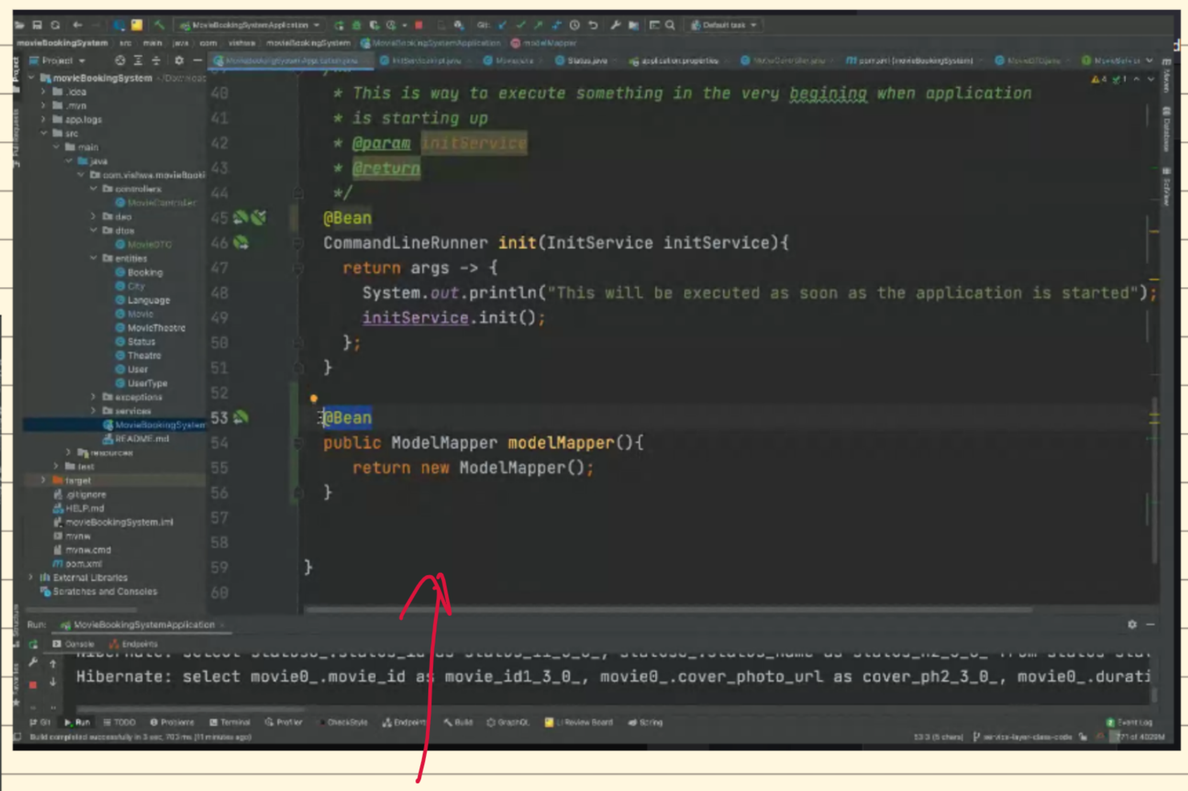


DTO (Data transfer object)

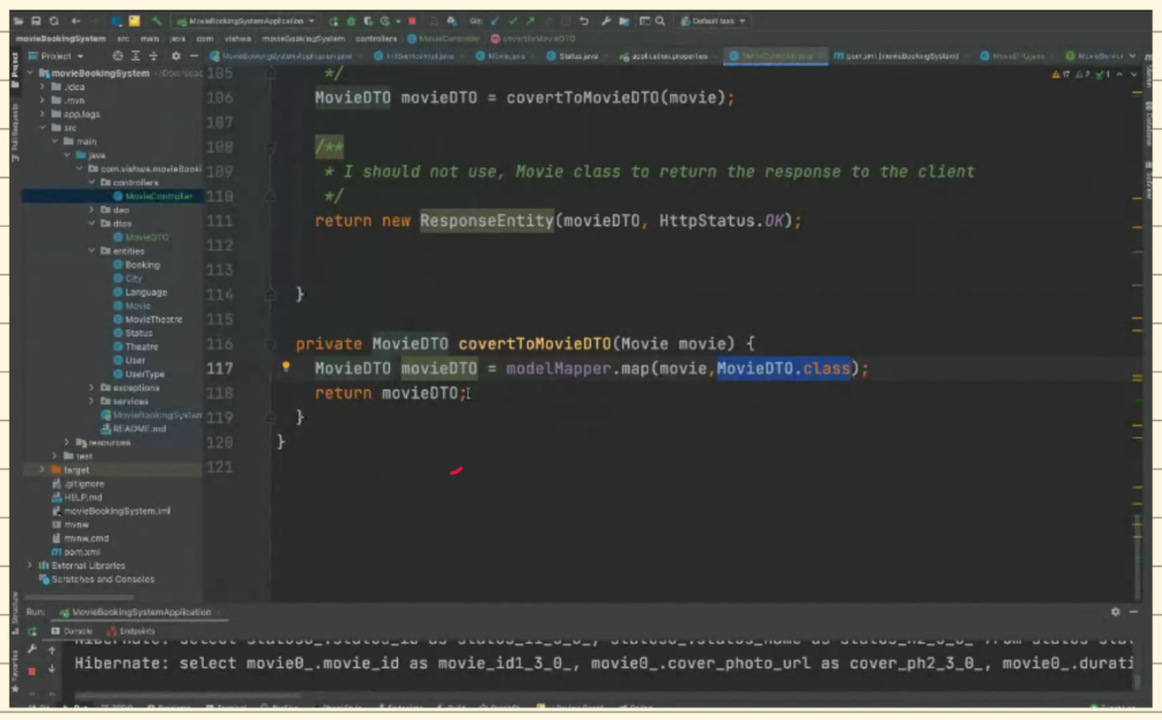
Model mapper (Create new class with some fields)



@Bean ~~ok~~ this is ok



Create a Bean of m



@JsonProperty("movie_name") to create a String movieName; class

to customize JSON Property

@PostMapping(consumes = MediaType.Application_JsonValue,
Produces = MediaType.Applic_JsonValue

Headers (1)

Key	Value
Content-Type	application/json

Header

```java
 * This method expects some request body
 *
 *
 * @param movie
 * @return
 */
@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE ,
    produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity  createMovie(@RequestBody MovieDTO movieDTO ){
    //Logic to create the movie
}

private MovieDTO covertToMovieDTO(Movie movie) {
  MovieDTO movieDTO = modelMapper.map(movie,MovieDTO.class);
  return movieDTO;
}
```

```java
 * @param movie
 * @return
 */
@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE ,
    produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<MovieDTO>  createMovie( MovieDTO movieDTO ){
   //Logic to create the movie
}

private MovieDTO covertToMovieDTO(Movie movie) {
  MovieDTO movieDTO = modelMapper.map(movie,MovieDTO.class);
  return movieDTO;
}
}
```

Convert Request Body to Movie DTO

```java
 *
 * @param movie
 * @return
 */
@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE ,
    produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity  createMovie(@RequestBody MovieDTO movieDTO ){
   //I need to create the new movie
  //I need to create Movie object from MovieDTO object
  Movie movie = modelMapper.map(movieDTO,Movie.class);

  Movie savedMovie = movieService.acceptMovieDetails(movie);

  //Again I need to covert it back to MovieDTO to send back to client

  MovieDTO responseBody = modelMapper.map(savedMovie,MovieDTO.class);

  return new ResponseEntity(responseBody, HttpStatus.CREATED);
}
20:02:01.102  INFO 13380 --- [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet        : Compl
```

@RequestBody
4 /{cart_Id}
@RequestParam ("cart_Id"

Optional

```java
 * @param movie
 * @return
 */
@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE ,
    produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity  createMovie( MovieDTO movieDTO ){
   //Logic to create the movie
}
```

Bad Request

```java
 * GET 127.0.0.1:8082:mbs/v1/movies/{movieId}
 *
 * GET 127.0.0.1:8082:mbs/v1/movies/10
 *
 */
@GetMapping("/{movieId}")
public ResponseEntity getMovieBasedOnId(
    @PathVariable(name="movieId") int movieId)  {
  Movie movie = null ;
  try {
    movie = movieService.getMovieDetails(movieId);
  } catch(MovieDetailsNotFoundException e){
      return new ResponseEntity( body: "movieId : [ "+ movieId + " ] passed is not correct",
          HttpStatus.BAD_REQUEST);
  }

  /**
   * Convert the Movie object to MovieDTO object
   */
```

handle

```java
 */
@GetMapping("/{movieId}")
public ResponseEntity getMovieBasedOnId(
    @PathVariable(name="movieId") int movieId)  {
  Movie movie = null ;
  try {
    movie = movieService.getMovieDetails(movieId);
  } catch(MovieDetailsNotFoundException e){
    LOGGER.error("Bad request found for the id : "+ movieId);
    LOGGER.error( s: "Error stack trace :"+ e.t);
      return new ResponseEntity( body: "mov
          HttpStatus.BAD_REQUEST);
  }

  /**
```

```
toString()                 String
getLocalizedMessage()      String
getMessage()               String
getStackTrace()            StackTraceElement[]
fillInStackTrace()         Throwable
getCause()                 Throwable
getSuppressed()            Throwable[]
initCause(Throwable cause) Throwable
```

Import "logger"

class {

logger;

}

We can also log in a file

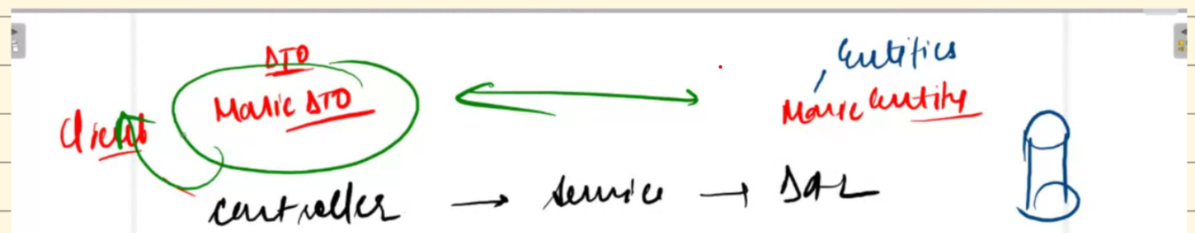
Controller → Service → DAL →

Json

Json object

Jackson library to achive this

@Entity ← is only for DB so we make a copy of By name ClassDTO



DTO (Data Transfer Object)

```java
import java.time.LocalDateTime;

/**
 * This class will be used to convert client JSON request to Java Object and
 * vice-versa
 */
public class MovieDTO {

  private int movieId ;
  private String movieName;
  private String movieDescription;
  private LocalDateTime releaseDate;
  private int duration;
  private String coverPhotoUrl;
  private String trailerUrl;
  private int statusId;

  public int getMovieId() {
    return movieId;
  }

  public void setMovieId(int movieId) {
    this.movieId = movieId;
```

Model mapper ← Create a identical object

Create Bean using method

```java
@Bean
public ModelMapper modelMapper(){
    return new ModelMapper();
}
```

```xml
<!-- https://mvnrepository.com/artifact/org.mo
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>2.3.5</version>
</dependency>
```

```java
@Autowired
private ModelMapper modelMapper ;
/**
```

Could not autowire. No

Using model mapper

```java
    }

    private MovieDTO covertToMovieDTO(Movie movie) {
        MovieDTO movieDTO = modelMapper.map(movie,MovieDTO.class);
        return movieDTO;
    }
}
```

same object different type

```java
 */
public class MovieDTO {

  private int movieId ;
  @JsonProperty("movie_name")
  private String movieName;
  @JsonProperty("movie_desc")
  private String movieDescription;
  private LocalDateTime releaseDate;
```

from Jackson library to convert JSON key

Put Mapping

```java
 */
@PutMapping(value = "/{movieId}")
public ResponseEntity updateMovieDetails(MovieDTO movieDTO ,int movieId ){

}
```

Delete Mapping

```java
 * @return
 */
@DeleteMapping(value="/{movie_id}")
public ResponseEntity deleteMovie( @PathVariable(name ="movie_id" ) int id) throws Movi
  movieService.deleteMovie(id);

  return new ResponseEntity<>( body: "DELETED", HttpStatus.OK);
}
```

Validation

Handle exception in controller

```java
blic ResponseEntity getMovieBasedOnId(
  @PathVariable(name="movieId") int movieId)  {
Movie movie = null ;
try {
  movie = movieService.getMovieDetails(movieId);
} catch(MovieDetailsNotFoundException e){
  LOGGER.error("Bad request found for the id : "+ mo);
    return new ResponseEntity( body: "movieId : [ "+ movieId + " ] passed is not correct",
        HttpStatus.BAD_REQUEST);
  }
```

↑ manually

AOP (Aspect Oriented Programming)

Project

- C_1 ———
- C_2 ———
- C_3 ———
- C_4 ———

SRP

logging needed

↓

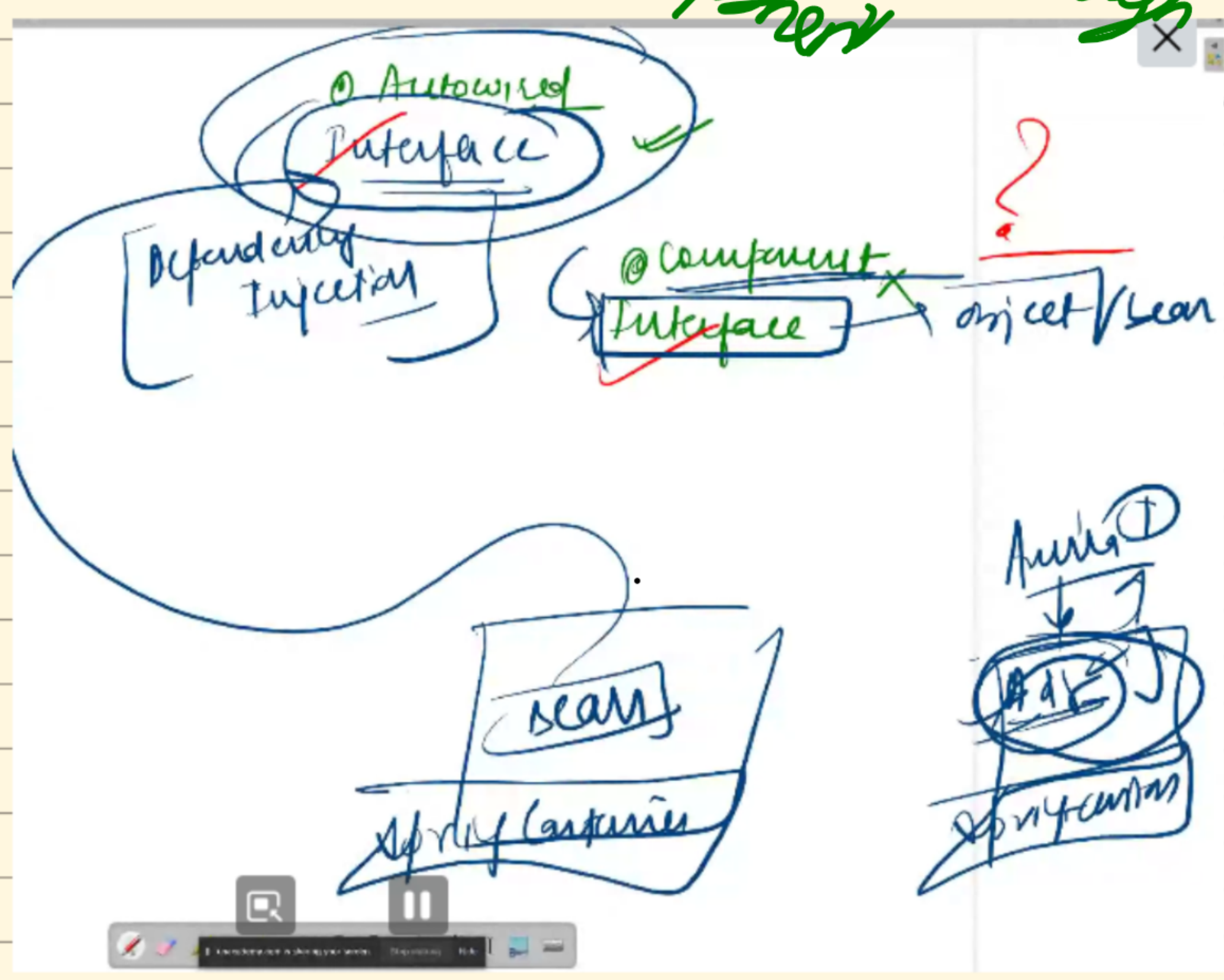
Cross cutting concern (use in all classes)

Logging is not functionally important (customer didn't requested for this)

AOP
↓ to achieve loosely couple component

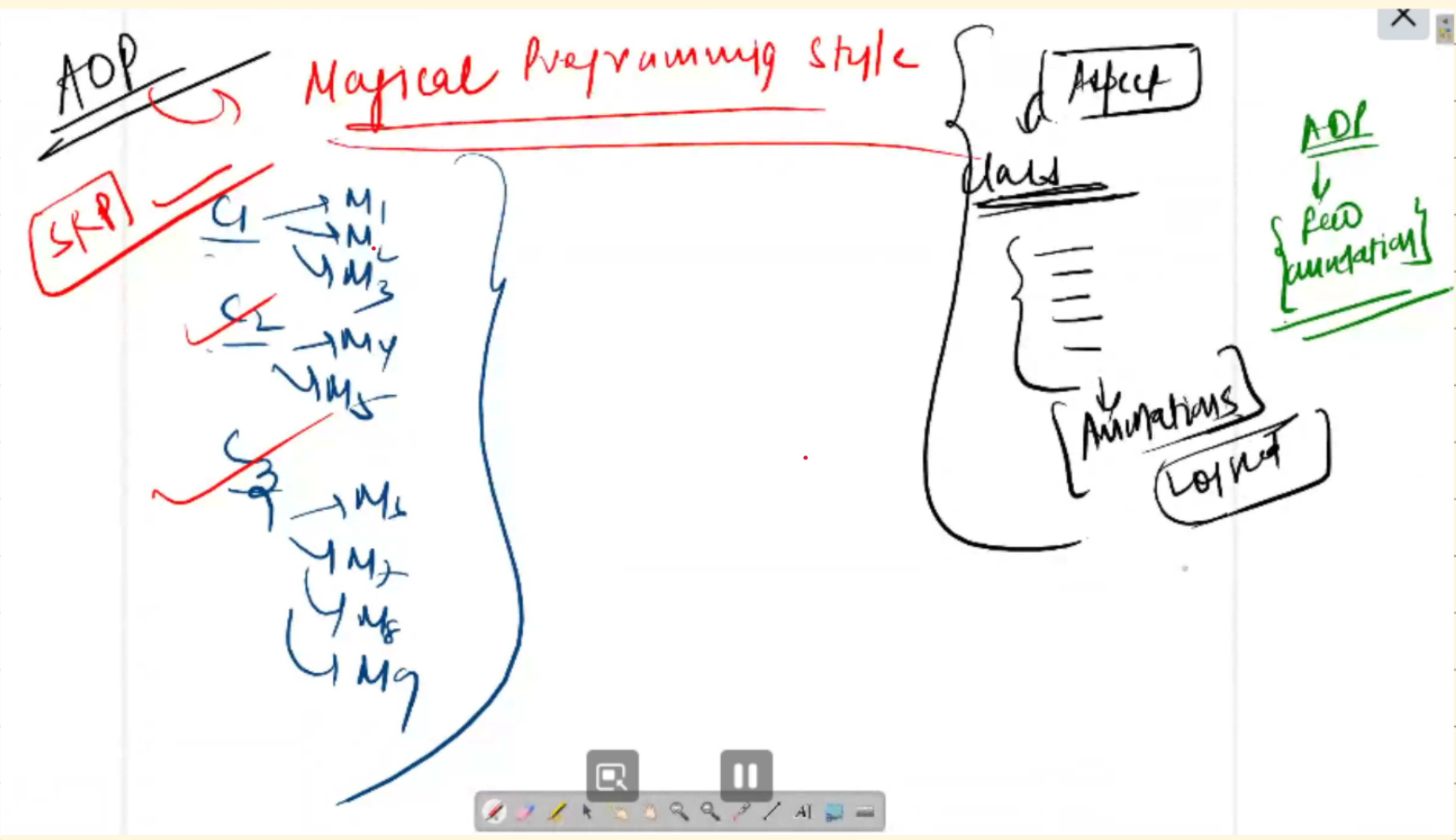
{ OOP
  function
  Procedure }

Programming paradigm



@component X interface

← because we can't create object from interface

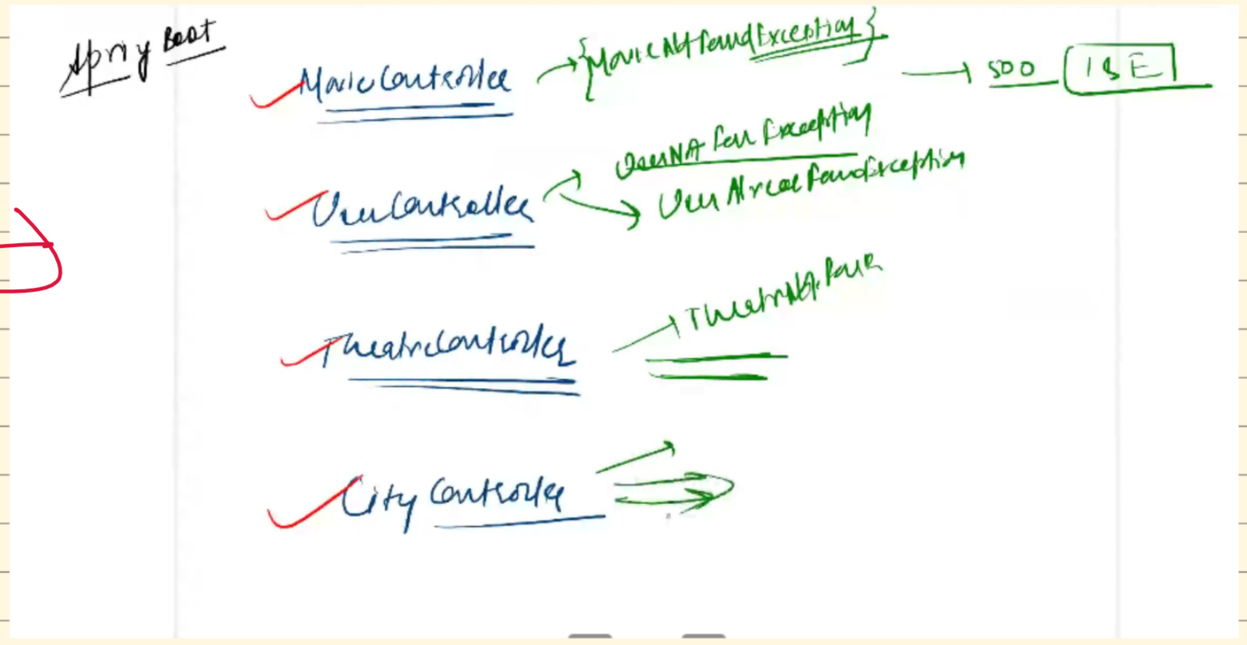


Any functionality which is cross cutting we use AOP

↓ use in many classes (and for changes we changing in this class)

#

exception by controller



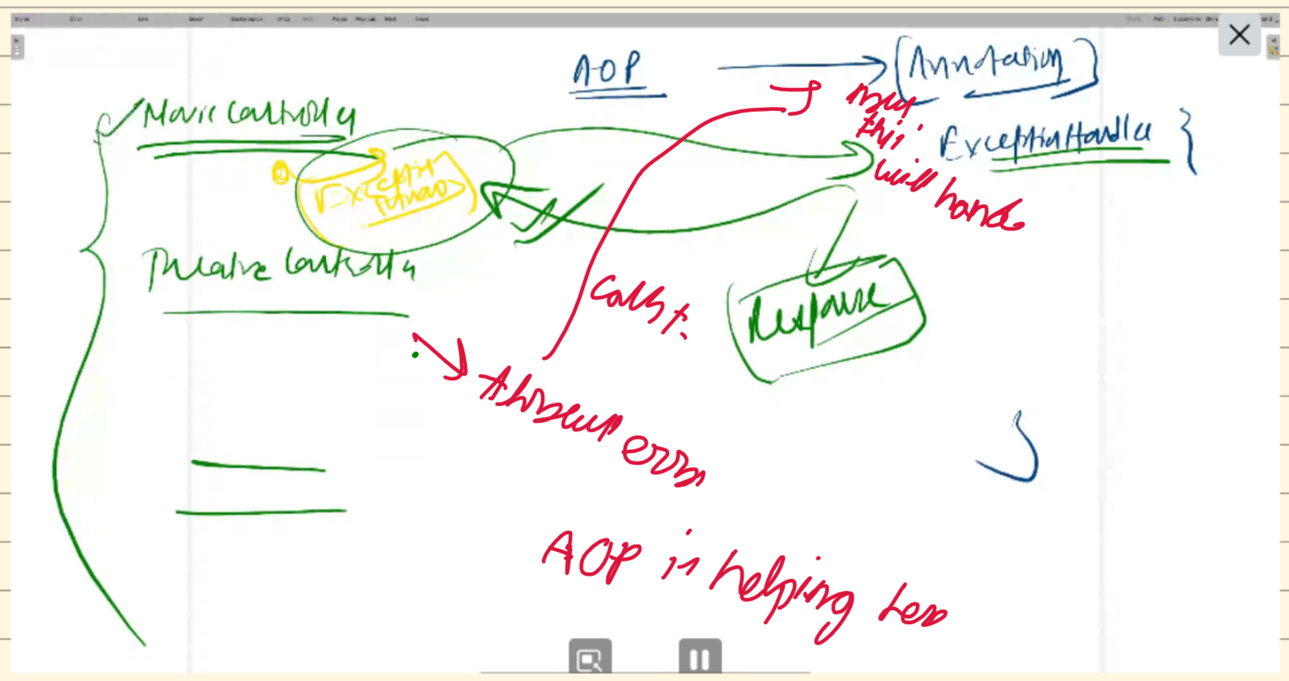
exception handler (using AOP)

@ControllerAdvice

only in Controller class

any exception in Controller layer

Calls the Calls and handle by this class




```java
@ControllerAdvice
/**
 * 1. Create the bean of this class and make it available in Spring Container
 * 2. Everytime an exception is thrown this class will be informed
 */
public class MovieExceptionHandler {
  /**
   * This annotation, indicate to this method, that if the given exception
   * happens, below method should be called.
   * @return
   */
  @ExceptionHandler(value= MovieDetailsNotFoundException.class)
  public ResponseEntity handleMovieNotFoundException(){

  }

}
```

```java
@ControllerAdvice
/**
 * 1. Create the bean of this class and make it available in Spring Container
 * 2. Everytime an exception is thrown this class will be informed
 */ public class MovieExceptionHandler {

  private static final Logger LOGGER = LoggerFactory.getLogger(MovieExceptionHandler.class);

  /**
   * This annotation, indicate to this method, that if the given exception
   * happens, below method should be called.
   * @return
   */
  @ExceptionHandler(value = MovieDetailsNotFoundException.class)
  public ResponseEntity handleMovieNotFoundException() {
    LOGGER.error("Exception happened , movie id is not available");
    return new ResponseEntity( body: "Movie ID passed is not available", HttpStatus.BAD_REQUEST);
  }
}
```