Kết nối cơ sở dữ liệu với JDBC nhằm thực hiện chức năng tìm kiếm

Bước 1: Tạo kết nối với CSDL

**static** **final** String ***DB\_URL***="jdbc:mysql://localhost:3306/estatebasic";

**static** **final** String ***USER***="root";

**static** **final** String ***PASS***="Hoanghiep2003@";

Bước 2: Tạo đối tượng Statement

+,xác định cậu lệnh SQL:

String sql ="SELECTE \* FROM building";

+,tạo đối tượng statement (Dùng để truy vấn có 3 phương thức là: executeQuery()//Truy van,executeUpdate() cap nhat,execute() truyvan hoac capnhat) + tạo Driver(Thành phần trung gian giữa ctr và csdl)

**try**(Connection conn=DriverManager.*getConnection*(***DB\_URL***,***USER***,***PASS***);

Statement stmt=conn.createStatement();

ResultSet rs=stmt.executeQuery(sql);){

//xử lí

}

Hoàn chỉnh:

**static** **final** String ***DB\_URL***="jdbc:mysql://localhost:3306/estatebasic";

**static** **final** String ***USER***="root";

**static** **final** String ***PASS***="Hoanghiep2003@";

**public** List<BuildingDTO> getBuilding(){

String sql ="SELECTE \* FROM building";

List<BuildingDTO> resultBuildingDTOs =**new** ArrayList<>();

**try**(Connection conn=DriverManager.*getConnection*(***DB\_URL***,***USER***,***PASS***);

Statement stmt=conn.createStatement();

ResultSet rs=stmt.executeQuery(sql);){

**while**(rs.next()) {

BuildingDTO building=**new** BuildingDTO();

}

}**catch**(SQLException e) {

e.printStackTrace();

System.***out***.print("Connected database failed...");

}

}

**-SOLID**: là viết tắt của 5 Rule

**1. Single responsibility principle**

-tức là Mỗi class chỉ nên thực hiện duy nhất 1 công việc

Đây là một ví dụ về lớp Java không tuân theo nguyên tắc trách nhiệm duy nhất (SRP):

public class Vehicle {  
    public void printDetails() {}  
    public double calculateValue() {}  
    public void addVehicleToDB() {}  
}

Lớp này Vehiclecó ba trách nhiệm riêng biệt: báo cáo, tính toán và cơ sở dữ liệu. Bằng cách áp dụng SRP, chúng ta có thể tách lớp trên thành ba lớp với các trách nhiệm riêng biệt.

Cái rule đầu tiên này có liên quan đến cái mô hình 3 layer và mvc bên dưới….

Còn nữa hãy tự tìm hiểu…

**mô hình 3 Layer Và MVC(Model-View-Controller):**

Ta cùng xem xét hàm getbuilding sau, đáng lẽ ra nó chỉ có mỗi 1 công việc là nhận data từ client nhưng ở đây nó làm cả nhận request , thao tác dưới DB, rồi trả ra dữ liệu theo yêu cầu của client

@GetMapping(value="/api/building/")

**public** List<BuildingDTO> getBuilding(@RequestParam(name="name",required = **false**)String name){

String sql ="SELECT \* FROM building b where name like '%"+name+"'%'";

List<BuildingDTO> result =**new** ArrayList<>();

**try**(Connection conn=DriverManager.*getConnection*(***DB\_URL***,***USER***,***PASS***);

Statement stmt=conn.createStatement();

ResultSet rs=stmt.executeQuery(sql);){

**while**(rs.next()) {

BuildingDTO building=**new** BuildingDTO();

building.setName(rs.getString("name"));

building.setStreet(rs.getString("street"));

building.setWard(rs.getString("ward"));

building.setNumberOfBasement(rs.getInt("numberofbasement"));

result.add(building);

}

}**catch**(SQLException e) {

e.printStackTrace();

System.***out***.print("Connected database failed...");

}

**return** result;

}

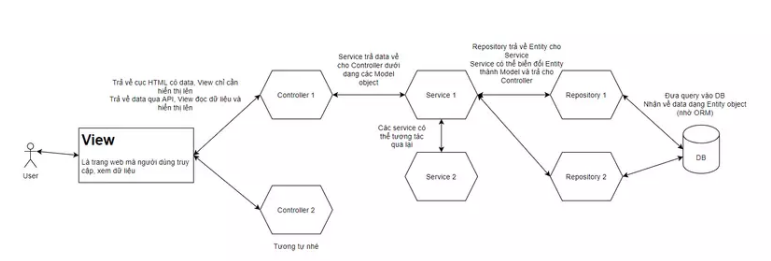
+ Code quá nhiều vi phạm Solid:không nên nhét quá nhiều code vào 1 class hay 1 hàm mà nên chia việc ra

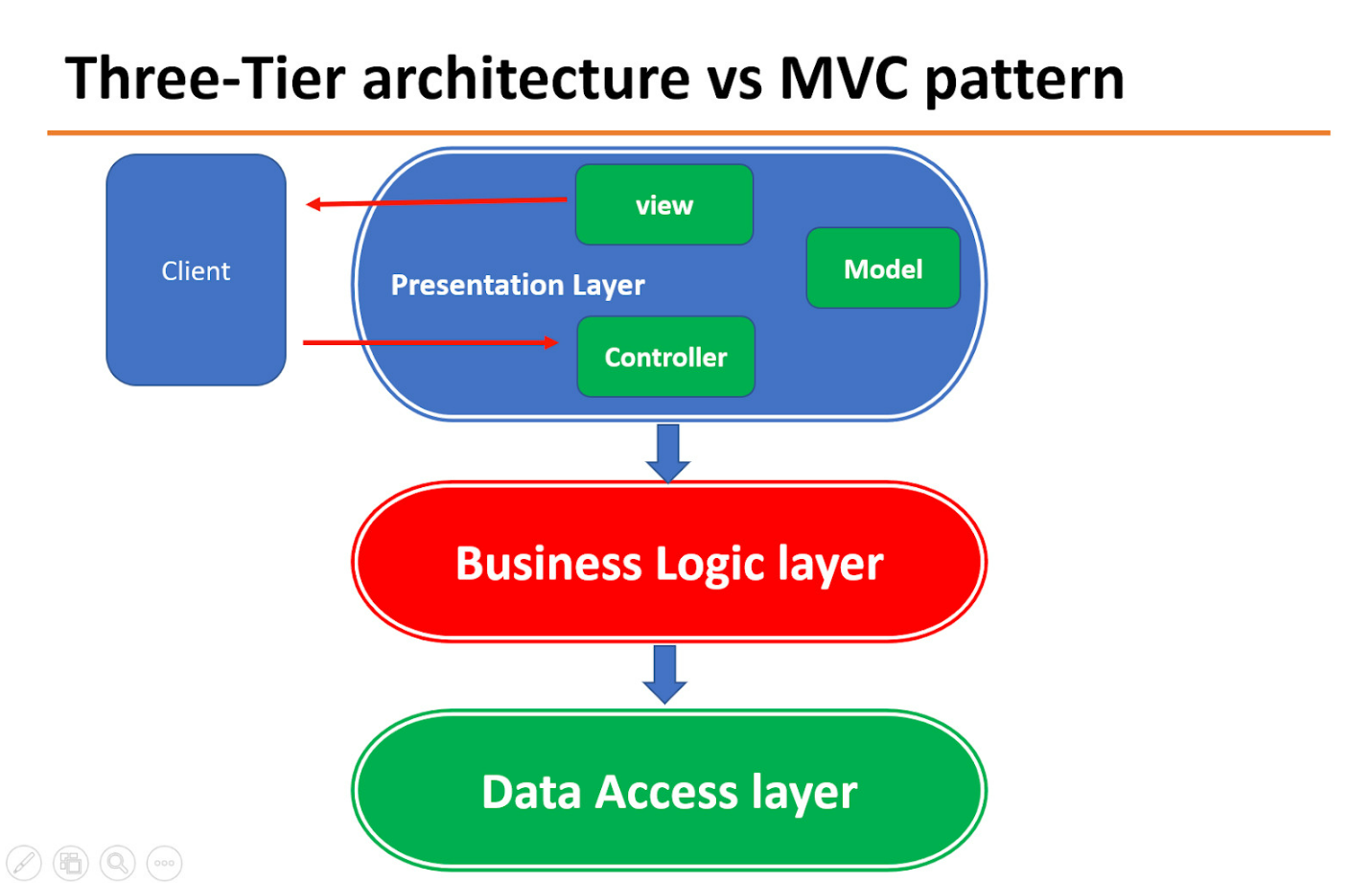
+ Người đến sau không hiểu gì cả, rất khó debug

=>Để khắc phục vấn đề trên ta sẽ dùng mô hình 3 Layer Và MVC

**Mô hình 3 lớp (3-layer architecture):** Tập trung vào việc chia ứng dụng thành ba lớp chính (Presentation Layer, Business Logic Layer, Data Access Layer). Mô hình này giúp tách biệt: giao diện người dùng, logic kinh doanh và xử lý dữ liệu.

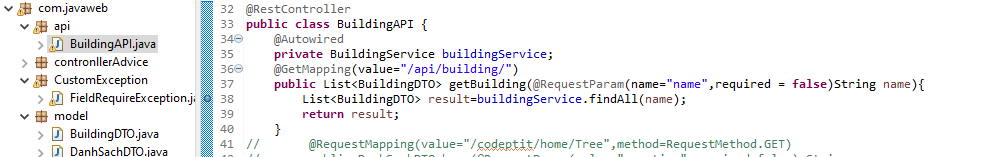
**MVC (Model-View-Controller):** Tách biệt ứng dụng thành ba thành phần chính (Model, View, Controller). Model(sau khi ta thao tác DB hết rồi thì cầm dữ liệu đó quăng ra cho client) đại diện cho dữ liệu và logic kinh doanh, View đại diện cho giao diện người dùng, và Controller quản lý luồng điều khiển và tương tác giữa Model và View.





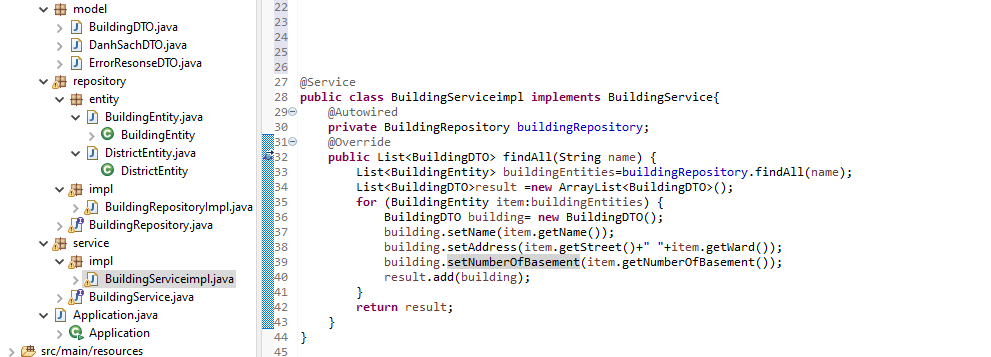
Tầng 1:Presentation Layer(Chứa MVC)=> @Restcontroller =>controller class thường tương ứng với phần View của MVC

-Tầng này chứa Anotation @RestController là nới tiếp nhận yêu cầu từ phía client



Tầng 2:Business Logic Layer =>@Service => service class xử lí dữ liệu đầu ra theo yêu cầu của client thường tương ứng với phần Model của MVC

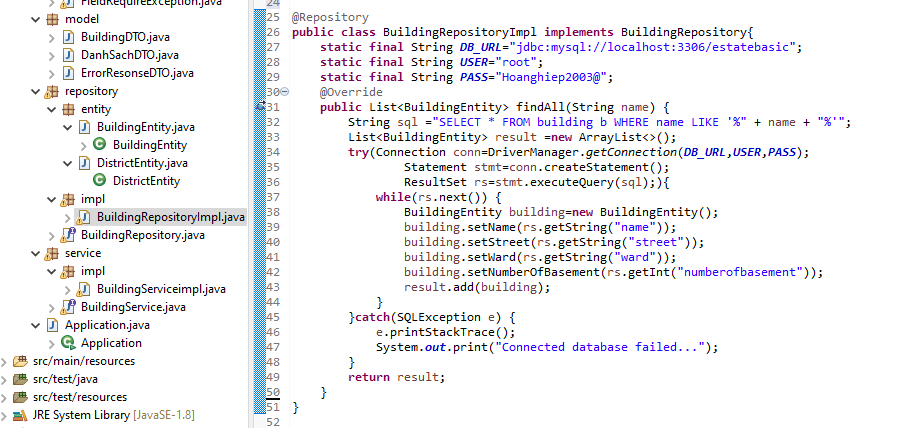
-Tầng này sẽ là nơi tiếp nhận dữ liệu trả về từ phía DB thông qua tầng 3 , rồi sau sẽ filter để lọc dữ liệu giống với bên client yêu cầu ,hãy nhìn pakage model ở đó chứa các class có nhiệm vụ trả dữ liệu đúng với phía yêu cầu ở bên client vậy nên sau khi filter xong ta sẽ trả về đối tượng DTO tương ứng.



Tầng 3:Data Access Layer =>@Repository =>repocitory class =>thao tác với DB, xử lí câu querry

Như ví dụ trên khi ta hứng dữ liệu từ phía db ta sử dụng DTO nhưng giờ ta không làm như vậy nữa ta sẽ dùng Entity (map 1:1 với DB*, map 1:1 ở đây là dữ liệu thằng DB có gì Entity sẽ tạo số field tương ứng*) còn DTO sẽ là nơi tả trả dữ liệu về sau khi ta filter xog

Công việc này nằm ở tầng 2.

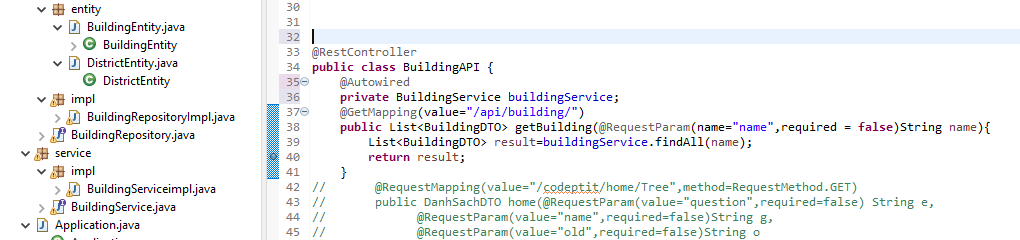


@Autowired (của Spring): hiểu đơn giản ,khi mộ class mà không có contructer ta khai báo đối tượng trực tiếp thì ta không thể nào mà sử dụng được phương thức của Class đó

Vd:SinhVien a (Nhìn đã thấy vô lí)

Mà bản thân thằng interface đã không có contructer vậy nên nếu khai báo thằng @Autowired trước khi khởi tạo interface hay bất kì thằng nào không có contructer nó sẽ tự động thêm contructe vào và sau đó ta có thể sử dụng các phương thức của nó.

**Bản thân tự** **tìm hiểu:** thằng @Autowired này nên dùng trong những trường hợp những thằng không có contructer, vd điển hình là interface thì nó là thằng có nhiều thằng class triển khai từ nó và nếu nhiều thằng cùng triển khai mà khi ta khai báo interface với không contructer nó sẽ không biết thằng method() này được khải triển bởi thằng nào => gây ra đa hình.

Ví dụ 

-thằng interface BuildingSevice có phương thức findAll() được triển khai từ thằng class BuildingSeviceimpl, dòng 39 đã sử dụng findAll() lúc này nó tự hiểu findAll() đươc triển khai từ class BuildingSeviceimpl bởi chỉ có duy nhất thằng này implements, vậy nếu có nhiều class implement từ thằng BuildingSevice thì hàm findALL() sẽ bị đa hình vì không biết cụ thể của thằng class nào.

=>cách khắc phục : private BuildingService buildingService = new BuildingServiceimpl();

đang sử dụng tính đa hình, và việc gán giá trị từ một lớp implement interface vào một biến có kiểu là interface là hoàn toàn hợp lệ trong Java.

Điều này sẽ giúp cho việc biết cụ thể interface được triển khai từ thằng nào mà không dùng đến @Autowired

Nhưng trong khóa học này theo tôi nghĩ là với mỗi một Entity thì sẽ sử dụng riêng một interface và 1 class triển khai inteface đó nên việc chỉ rõ có thể cần hoặc không.

1.Trong lớp **BuildingAPI**, bạn có một trường **buildingService** được đánh dấu bằng **@Autowired**, do đó Spring sẽ tự động chèn (inject) một đối tượng **BuildingService** đã được quản lý vào trường này.

2.Khi bạn gọi **buildingService.findAll(name)**, thực tế là bạn đang gọi phương thức **findAll** của đối tượng **BuildingServiceImpl** (vì **BuildingServiceImpl** triển khai **BuildingService**).

3.Trong phương thức **findAll** của **BuildingServiceImpl**, nó sử dụng **BuildingRepositoryimpl** để lấy danh sách **BuildingEntity** từ cơ sở dữ liệu dựa trên tên.

4.Sau đó, nó chuyển đổi danh sách **BuildingEntity** thành danh sách **BuildingDTO** bằng cách tạo mới các đối tượng **BuildingDTO** và đổ dữ liệu từ **BuildingEntity** sang **BuildingDTO**.

5.Cuối cùng, danh sách **BuildingDTO** được trả về từ phương thức **findAll** của **BuildingServiceImpl** và sau đó được gán cho biến **result** trong lớp **BuildingAPI**.

Lấy dữ liệu:

Tổng quát :Resultset->(java bean)Entity (Map 1:1 với DB)->filter

->DTO(model)->view

Update dữ liệu:

View->DTO(model)->Filter->Entity->DB

=>Kể từ này về sau Phải code theo mô hình 3 layer và MVC để code trở nên tối ưu và dễ phát triển

Xây dựng mô hình MVC,3-layer

Tip: kể hàm cũng vậy trong quyển clean code có nói một hầm không nên có quá 3 tham số nếu có hãy giảm số lượng biến xuống.

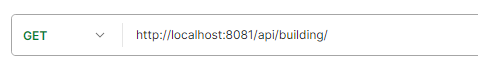
Robert C. Martin khuyến khích rằng một hàm (hoặc phương thức) không nên có quá 2 hoặc 3 tham số. Ông cho rằng nếu có quá nhiều tham số, hàm sẽ trở nên khó hiểu và khó duyệt, gây khó khăn trong việc bảo trì và sửa lỗi.

Thay vào đó, ông đề xuất cố gắng giảm số lượng tham số bằng cách sử dụng các cấu trúc dữ liệu (ví dụ như các đối tượng) để gom nhóm thông tin liên quan lại với nhau. Việc này giúp tăng tính nhất quán và dễ đọc của mã nguồn.

=>Thực chất solid hay Mô hình 3 layer MVC cũng chính là một trong những Design Pattens để giúp cho chúng ta Code quy củ và Clean hơn…

-Tùy vào cái yêu cầu của khách hàng ta sẽ trả về data theo yêu cầu:

Như sau đây không có yêu cầu gì, nghĩa là khách hàng sẽ tìm tất cả tòa nhà.



Như này là trả về tên những tòa nhà có tên là abc.

