# Technical Report

# Smart Sales

## 1. Giới thiệu

Smart Sales là ứng dụng web được thiết kế dựa theo CRM (Customer Relationship Management) để sử dụng cho mục đích chăm sóc khách hàng và tối ưu hoá quản lý bán hàng. Được xây dựng bằng Spring Boot với Java 21 và kiến trúc MVC, Smart Sales hỗ trợ các tính năng với khách hàng như: đăng kí, đăng nhập, mua sản phẩm, gửi yêu cầu hỗ trợ, trong khi đó với nhà quản lý có thể quản lý sản phẩm, quản lý khách hàng và giám sát các yêu cầu từ phía khách hàng. Dự án sử dụng một số công nghệ như Spring Boot, SpringMVC, Spring Data JPA, Hibernate, Spring Security, Bootstrap 5 cùng với template engine là Thymeleaf.

## 2. Mẫu kiến trúc

### 2.1. Kiến trúc MVC

Kiến trúc MVC (Model – View – Controller) là mô hình kiến trúc phần mềm được sử dụng phổ biến, trong đó chia phần mềm ra làm 3 phần riêng biệt:

- Model: Quản lý dữ liệu, tương tác với cơ sở dữ liệu và xử lý dữ liệu trả về cho View và Controller.
- View: Giao diện người dùng, chịu trách nhiệm hiển thị dữ liệu từ model và nhận đầu vào từ người dùng.
- Controller: Trung gian giữa View và Model. Nhận các yêu cầu người dùng từ View, xử lý các yêu cầu (có thể thông qua Model) và trả lại View tương ứng

### 2.2. Kiến trúc 3 lớp

Kiến trúc 3 lớp là kiến trúc được xây dựng theo client/server, trong đó được chia ra làm 3 lớp chính:

- Tầng presentation (UI/Controller): Xử lý giao diện người dùng và điều hướng các yêu cầu tới các dịch vụ phù hợp.
- Tầng business (Service): Xử lý các nghiệp vụ tính toán của phần mềm và xử lý thông tin giữa 2 tầng trên dưới.
- Tầng Data Access (Repository): Quản lý việc lưu trữ dữ liệu và truy vấn dữ liệu.

Việc sử dụng kiến trúc 3 lớp giúp nâng cao tính module, tăng cường khả năng bảo trì và mở rộng dự án.

2.3. DTO (Data Transfer Object)

DTO được sử dụng để chuyển dữ liệu thông qua các tầng (chủ yếu là controller và service) mà không cần chuyển trực tiếp các đối tượng của dữ liệu (entity). Việc này cho phép người dùng chỉ tương tác với các dữ liệu cần thiết của một đối tượng thay vì tương tác trực tiếp với đối tượng đó.

3. Các tính năng của dự án

3.1. Authentication và authorization:

- Sử dụng Spring Security để cho phép người dùng có thể đăng nhập, đăng kí hệ thống cũng như truy cập vào các tính năng được cấp phép.

3.2. Các tính năng khách hàng:

- Xem các sản phẩm được đăng bán
- Mua hàng và sinh hoá đơn
- Xem lịch sử hoá đơn
- Yêu cầu chăm sóc khách hàng

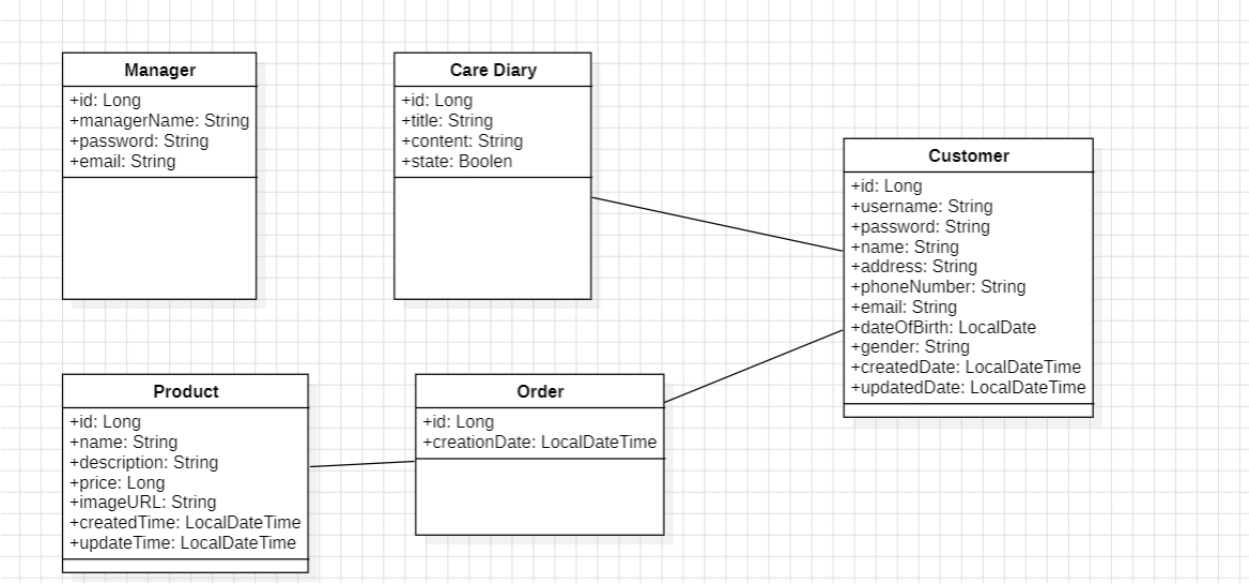
3.3. Các tính năng cho người quản lý:

- Quản lý sản phẩm (Thêm, sửa, xoá)

- Xem các đơn hàng của khách
- Quản lý khách hàng
- Hỗ trợ khách hàng khi có yêu cầu chăm sóc

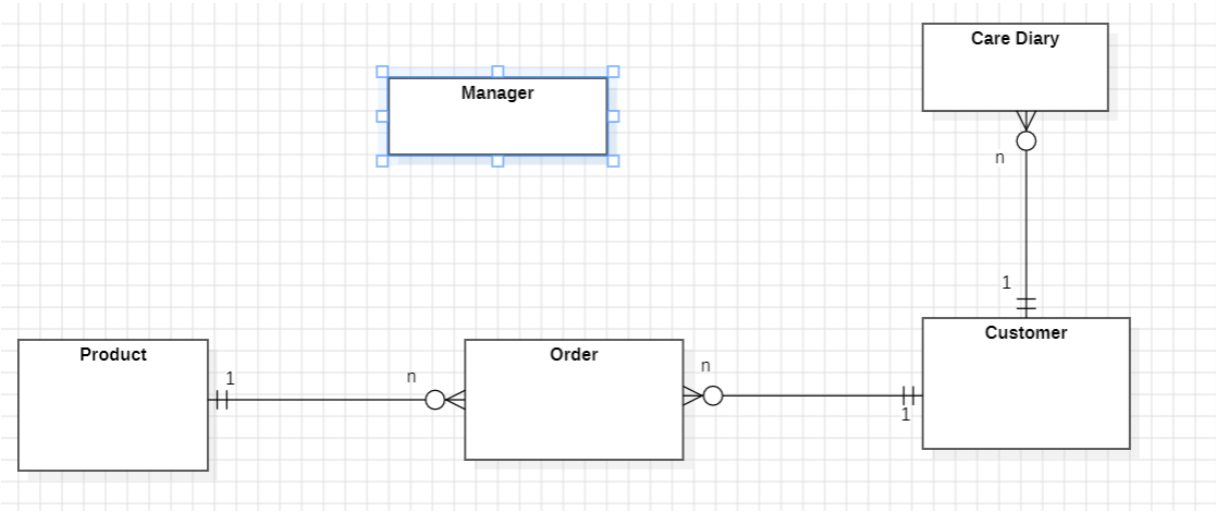
# 4. Thiết kế Database

## 4.1. Biểu đồ lớp




Biểu đồ lớp dùng để mô tả các thuộc tính và các phương thức của một class và thể hiện các mối quan hệ của các class đó. Kết hợp biểu đồ lớp và biểu đồ ER có thể giúp ta tạo ra cơ sở dữ liệu.

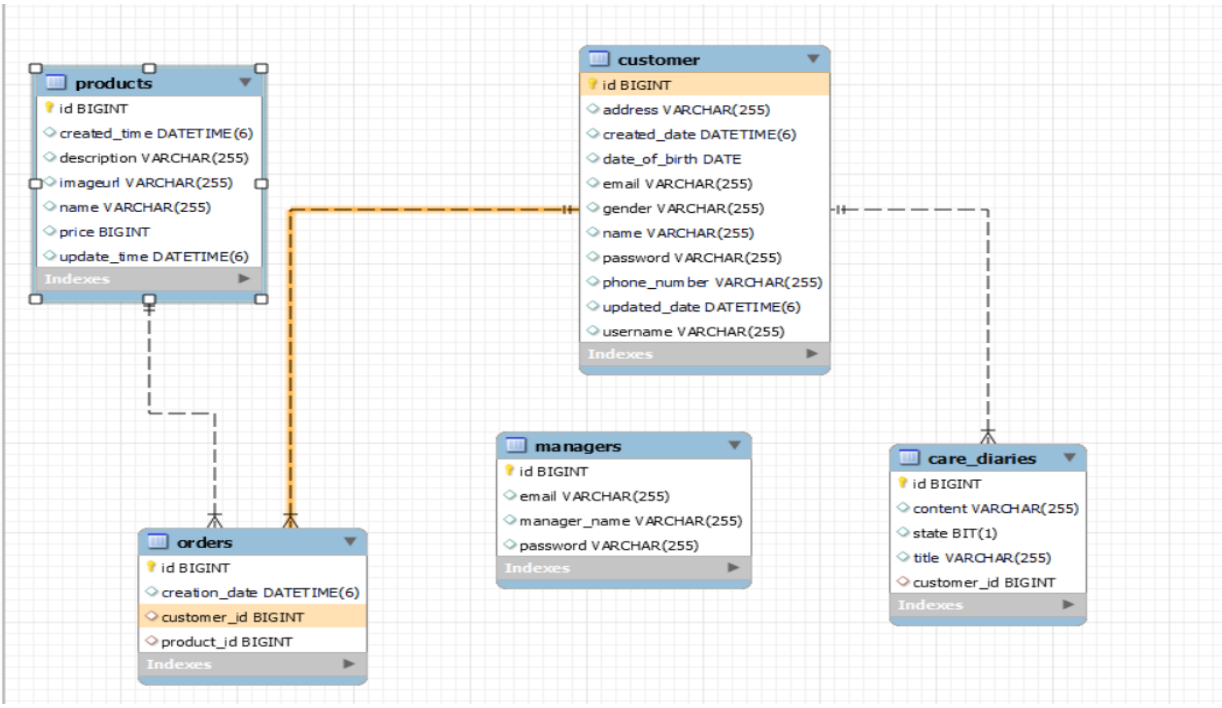
## 4.2. Biểu đồ ER



Biểu đồ này thể hiện mối quan hệ giữa các thực thể thông qua các liên kết. Kết hợp biểu đồ này và biểu đồ lớp ta có thể tạo một cơ sở dữ liệu.

4.3. Biểu đồ cơ sở dữ liệu sau khi xây dựng

Sau khi xây dựng xong cơ sở biểu đồ lớp và biểu đồ ER, chúng ta có thể xây dựng cơ sở dữ liệu. Dưới đây là biểu đồ cơ sở dữ liệu sau khi được xây dựng:

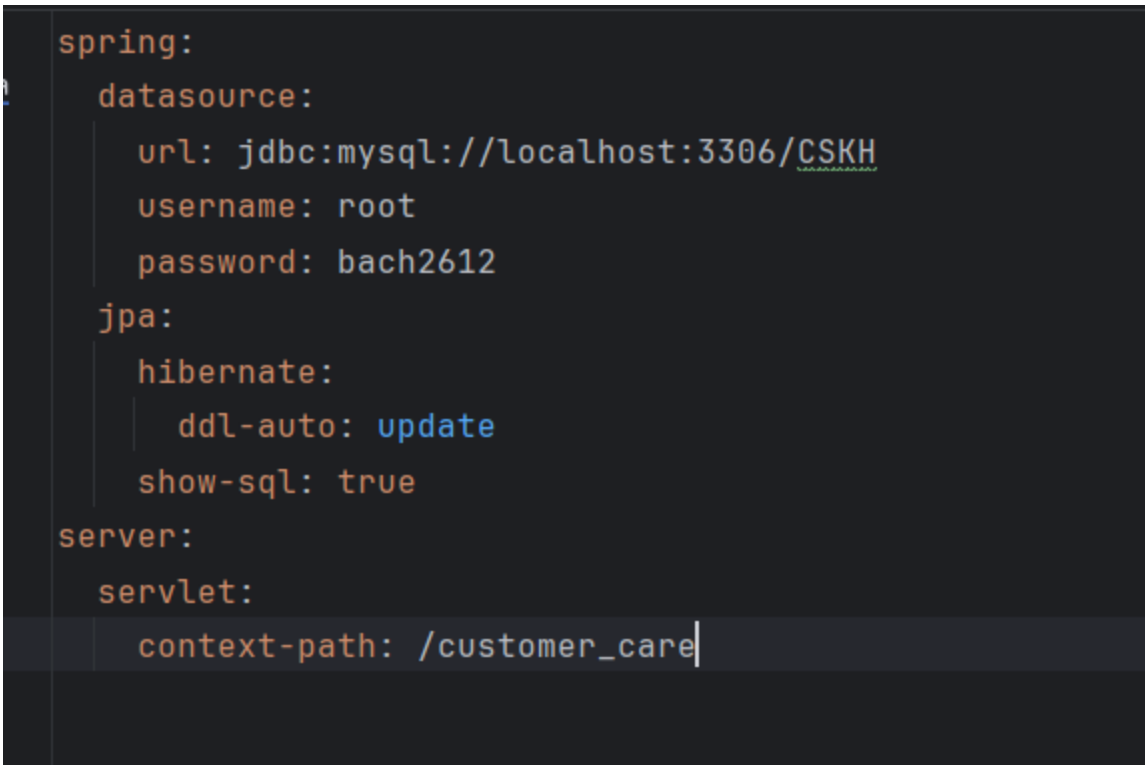



5. Triển khai dự án

Do hạn chế về mặt thời gian, dưới đây tôi sẽ chỉ mô tả một số đoạn code chính và hạn chế các đoạn code có nội dung tương tự để có thể tối ưu số phần mà tôi có thể demo.

5.1. Thiết lập settings cho dự án trong file application.yaml

Tôi sử dụng file yaml để phục vụ cho cấu hình dự án. Dưới đây là cấu hình cho dự án mà tôi đã sử dụng.




- Với spring/datasource, tôi thiết lập url đến server của database, cụ thể là MySQL và username, password cho server của tôi.
- Với spring/jpa tôi thiết lập cơ chế cho hibernate là auto update mỗi khi tôi sửa đổi code liên quan tới cơ sở dữ liệu, cùng với đó là show ra các câu lệnh sql mỗi khi tương tác với cơ sở dữ liệu.
- Với server/servlet, tôi thiết lập URL mặc định cho dự án là /customer_care

## 5.2. Xây dựng Model:

Dựa vào biểu đồ lớp và biểu đồ ER kết hợp với Spring Data JPA ta tạo ra các class chứa các thuộc tính trong các biểu đồ trên. Dưới đây là demo class Care Diary sau khi được triển khai dự án:

```
@Entity  17 usages  lyduybach2612
@Table(name = "care_diaries")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@FieldDefaults(level = AccessLevel.PRIVATE)
public class CareDiary {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long id;
    String title;
    String content;
    Boolean state;
    @ManyToOne
    @JoinColumn(name = "customer_id")
    Customer customer;

}
```

- Các annotation @Entity và @Table cho phép khai báo một class là một thực thể trong cơ sở dữ liệu, điều này sẽ giúp chương trình khi chạy tạo ra một cơ sở dữ liệu thông qua cơ chế của Hibernate.
- Với Lombok, tôi có thể rút gọn các đoạn code dài dòng như get, set và các constructor bằng các annotation: @Data, @Builder, @NoArgsConstructor, @AllArgsConstructor.
- Sau khi tạo các trường dữ liệu, tôi sử dụng @Id cho thuộc tính id nhằm thông báo cho chương trình biết đây sẽ là

khoá chính cho bảng care_diaries trong cơ sở dữ liệu. Bên cạnh đó, tôi sử dụng @GeneratedValue để có thể tự sinh khoá chính khi không được cung cấp trước. Với strategy = GenerationType.IDENTITY, các khoá chính sẽ được tự sinh với các số thứ tự tăng dần bắt đầu từ 1.

- Với thuộc tính customer là thuộc tính được sinh ra do liên kết 1 – n từ customer và care_diary như trong biểu đồ ER. Với một mối quan hệ association trong biểu đồ lớp, tôi hoàn toàn có thể tự tạo một mối quan hệ bất kì vì association không chỉ ra một mối quan hệ nào nhất định. Với @ManyToOne và @ JoinColumn, chương trình khi chạy sẽ sinh ra một thuộc tính trong bảng care_diaries một cột khoá ngoại có tên là customer_id được kết nối từ bảng customers.

5.3. Xây dựng Repository cho các thực thể

Repository là tầng cho phép chúng ta truy vấn và tương tác trực tiếp với cơ sở dữ liệu. Tôi tạo ra một package có tên repository, sau đó tiến hành tạo các interface cho từng thực thể. Dưới đây là minh hoạ cho một thực thể.


Spring Data JPA cung cấp cho chúng ta interface JpaRepository chứa đựng các phương thức giúp chúng ta có

khả năng truy vấn các dữ liệu trong cơ sở dữ liệu và sử dụng JPQL để custom những câu query riêng.

- Trong đoạn code trên, tôi tạo ra một interface CareDiaryRepository extends (kế thừa) JpaRepository với CareDiary là tên class của thực thể và Long là kiểu dữ liệu của ID trong class CareDiary.
- JpaRepository sẽ cung cấp một lớp mặc định nhằm phục vụ implement interface cho chúng ta. Chính vì vậy, tôi không tạo một class để implement interface CareDiaryRepository.

## 5.4. Xây dựng Service cho các thực thể

Service là tầng chịu trách nhiệm xử lý các nghiệp vụ tính toán và vận chuyển dữ liệu cho các tầng trên dưới. Trước tiên, tôi tạo một package service để tạo ra các interface cho từng thực thể. Trong package này, tôi tạo thêm một package nữa là impl nhằm cung cấp các class implement lại interface service. Dưới đây là minh hoạ cho tầng service.

```
package com.bach.qlkh.service;

import ...

public interface CareDiaryService {  4 usages  1 implementation  lyduybach2612
    List<CareDiaryDto> getAllCareDiaries();  1 usage  1 implementation  lyduybach2612

    void createCareDiary(CareDiaryDto careDiary);  1 usage  1 implementation  lyduybach2612

    void activeCareDiary(Long careDiaryId);  1 usage  1 implementation  lyduybach2612

    CareDiaryDto findCareDiary(Long careDiaryId);  1 usage  1 implementation  lyduybach2612

    void updateCareDiary(Long careDiaryId, CareDiaryDto careDiary);  1 usage  1 implementation  lyduybach2612
}
```

- Đầu tiên tôi tạo ra một interface CareDiaryService và chứa các phương thức tôi muốn thực hiện. Về chi tiết các phương thức, tôi sẽ nói ở phần sau.
- Sau khi tạo interface, tôi tiến hành tạo một class implement lại interface đó.

```
@Service  lyduybach2612
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
@RequiredArgsConstructor
public class CareDiaryServiceImpl implements CareDiaryService {

    CareDiaryRepository careDiaryRepository;

    @Override  1 usage  lyduybach2612
    public List<CareDiaryDto> getAllCareDiaries() {

        List<CareDiary> careDiaries = careDiaryRepository.findAllByOrderByStateAsc();
        return careDiaries.stream().map(CareDiaryMapper::mapToCareDiaryDto).toList();

    }
```

- Trong class trên, tôi sử dụng annotation @Service để khai báo cho chương trình biết class này thuộc tầng service. Với annotation @FieldDefaults, tôi thiết lập các truy cập mặc định cho các thuộc tính là private và khởi tạo chúng là các biến final. Cuối cùng là annotation @RequiredArgsConstructor, nó giúp tôi tạo ra một constructor chứa các tham số là các thuộc tính kiểu final.
- Sau khi sử dụng các annotation trên, tôi khai báo một đối tượng CareDiaryRepository là một thuộc tính của class này. Nhờ vào Lombok, tôi sẽ được tạo một constructor, trong constructor đó tôi sẽ tiêm Dependency vào trong class thông qua Constructor Injection.

5.5. Xây dựng Controller cho các thực thể

Controller là tầng xử lý giao diện và điều hướng các HTTP request. Tôi tiến hành tạo một package mang tên controller để chứa toàn bộ các controller cho các thực thể. Sau đó tạo các class controller tương ứng trong package đó. Dưới đây là demo minh hoạ cho việc tạo một class controller.

```
@Controller  lyduybach2612
@RequestMapping("/careDiaries")
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
@RequiredArgsConstructor
public class CareDiaryController {

    CareDiaryService careDiaryService;
```

- Tương tự với service, tôi sử dụng 2 annotation cho việc khởi tạo mặc định access level và final cho các thuộc tính cũng như constructor cho việc Dependency Injection.
- Bên cạnh đó, tôi sử dụng thêm annotation @Controller để thông báo cho chương trình biết đây là class thuộc tầng controller và hỗ trợ trả các phương thức về view tương ứng. Cuối cùng là annotation @RequestMapping là annotation hỗ trợ tôi thiết lập các URL request mặc định là /careDiaries cho các phương thức ở trong controller.

5.6. Xây dựng các DTO và Mapper

- DTO (Data Transfer Object) là các class chỉ chứa dữ liệu nhằm phục vụ cho mục đích chuyển dữ liệu giữa các tầng controller và service. Qua đó, có thể đưa tới người dùng các dữ liệu cần thiết. Dưới đây là code minh hoạ.

```java
@Data  22 usages  lyduybach2612
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class CareDiaryDto {

    Long id;
    @NotEmpty(message = "Vấn đề đang gặp không được phép bỏ trống")
    String title;
    @NotEmpty(message = "Nội dung vấn đề không được phép bỏ trống")
    String content;
    Boolean state;
    Customer customer;

}
```

- Như đã nói ở trên, DTO chỉ đơn thuần là chứa dữ liệu, nên chúng ta chỉ khai báo các dữ liệu cần thiết cho class này và sử dụng các annotation để hỗ trợ cho việc tạo setter, getter cũng như các constructor cần thiết.
- Bên cạnh đó tôi cũng sử dụng các annotation liên quan tới Validation dữ liệu, để hỗ trợ chức năng Create và Update.
- Sau khi tạo các class DTO, tôi tiến hành tạo ra class Mapper tương ứng với các class đó. Trong mapper chứa các phương thức tĩnh (static) nhằm phục vụ chuyển đổi dữ liệu từ class thường sang DTO.

## 5.7. Xây dựng các tính năng CRUD

### 5.7.1. Select

Với tính năng đầu tiên, tương ứng với câu lệnh select, tôi sẽ show ra một list các nhật kí chăm sóc khách hàng. Dưới đây là code demo.

```
@GetMapping()  lyduybach2612 *
public String careDiaryList(Model model) {

    List<CareDiaryDto> careDiaries = careDiaryService.getAllCareDiaries();
    model.addAttribute( attributeName: "careDiaries", careDiaries);
    return "care-diary-list";

}
```

- Đầu tiên tôi tạo một phương thức với kiểu trả về là String. Trong phương thức đó tôi sử dụng annotation @Getmapping và không truyền đầu vào để sử dụng URL mặc định là /careDiaries. Tôi sử dụng một tham số là Model để thêm dữ liệu vào mô hình và trả dữ liệu về view. Trong phương thức này, tôi trả về một View có tên file là care-diary-list. Đây là file html chứa code giao diện.
- Tôi tạo ra một List<CareDiaryDto>, sau đó gọi đến service để lấy dữ liệu từ Repository. Sau khi lấy xong dữ liệu, tôi thêm dữ liệu đó về view để view xử lý.

```
@Override  1 usage  lyduybach2612
public List<CareDiaryDto> getAllCareDiaries() {

    List<CareDiary> careDiaries = careDiaryRepository.findAllByOrderByStateAsc();
    return careDiaries.stream().map(CareDiaryMapper::mapToCareDiaryDto).toList();

}
```

- Trong Service, tôi gọi tới Repository để lấy dữ liệu từ cơ sở dữ liệu và Map dữ liệu về DTO.

5.7.2. Create

Với tính năng Create, tôi tiến hành tạo ra các chức năng tạo ra các đối tượng cho các thực thể tương ứng. Dưới đây là code demo.

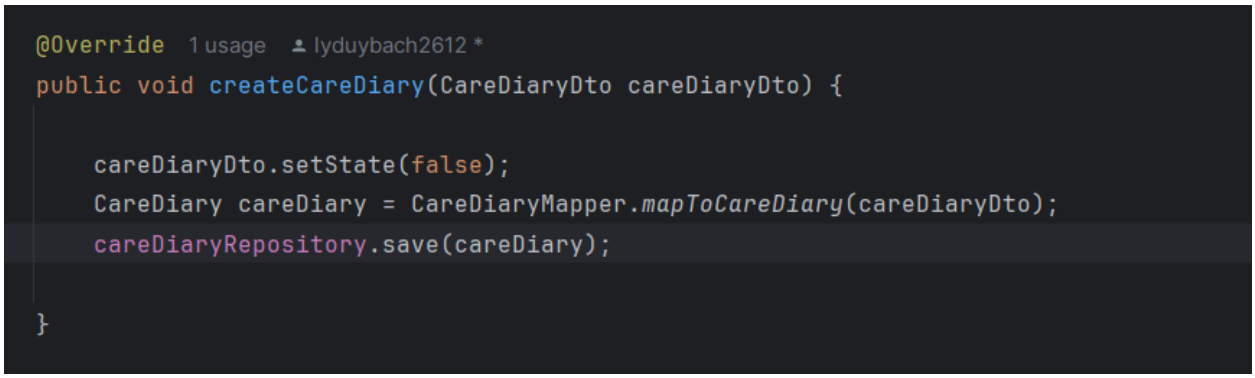
```
@GetMapping("/new")  lyduybach2612 *
public String createCareDiaryForm(Model model) {

    CareDiaryDto careDiary = new CareDiaryDto();
    model.addAttribute( attributeName: "careDiary", careDiary);
    return "create-care-diary";

}
```

- Đầu tiên tôi tiến hành tạo một phương thức get với URL /new. Trong phương thức này, tôi tạo ra một đối tượng thuộc class CareDiaryDto, sau đó trả nó về View.

```
@PostMapping("/new")  lyduybach2612
public String createCareDiary(@ModelAttribute("careDiary") @Valid CareDiaryDto careDiary,
                              BindingResult bindingResult, Model model) {

    if (bindingResult.hasErrors()) {
        model.addAttribute( attributeName: "careDiary", careDiary);
        return "create-care-diary";
    }

    careDiaryService.createCareDiary(careDiary);
    return "redirect:/careDiaries";

}
```

- Tiếp theo đó tôi tạo một phương thức post để gửi dữ liệu hỗ trợ cho việc tạo mới dữ liệu. Trong phương thức này tôi sử dụng model attribute đã thêm vào view ở trên cùng

với annotation @Valid để validation dữ liệu. Với tham số BindingResult cho phép tôi trả về lỗi của dữ liệu nếu có. Nếu bindingResult có lỗi, tôi sẽ add lại attribute về view sau đó trả lại trang hiện tại. Ngược lại nếu không có lỗi, tôi sẽ tiến hành gọi đến Service để dùng Repository lưu dữ liệu vào cơ sở dữ liệu. Sau đó, redirect lại về trang list nhật kí chăm sóc.



```
@Override  1 usage  lyduybach2612 *
public void createCareDiary(CareDiaryDto careDiaryDto) {

    careDiaryDto.setState(false);
    CareDiary careDiary = CareDiaryMapper.mapToCareDiary(careDiaryDto);
    careDiaryRepository.save(careDiary);

}
```

- Trong service, tôi set trạng thái ban đầu là false, sau đó Map DTO về dữ liệu gốc và gọi đến Repository để lưu dữ liệu lại.

5.7.3. Update

- Với tính năng Update, tôi tiến hành tạo ra một phương thức get để trả về view và một phương thức post để tiến hành thực thi việc cập nhập dữ liệu. Dưới đây là code demo.

```java
@PostMapping("/edit/{careDiaryId}") lyduybach2612
public String editCareDiary(@PathVariable("careDiaryId") Long careDiaryId,
                            @ModelAttribute("careDiary") @Valid CareDiaryDto careDiary,
                            BindingResult bindingResult, Model model) {

    if (bindingResult.hasErrors()) {
        model.addAttribute( attributeName: "careDiary", careDiary);
        return "edit-care-diary";
    }
    careDiaryService.updateCareDiary(careDiaryId,careDiary);
    return "redirect:/careDiaries";

}
```

```java
@GetMapping("/edit/{careDiaryId}") lyduybach2612 *
public String editCareDiaryForm(@PathVariable Long careDiaryId, Model model) {

    CareDiaryDto careDiary = careDiaryService.findCareDiary(careDiaryId);
    model.addAttribute( attributeName: "careDiary", careDiary);
    return "edit-care-diary";

}
```

```java
@Override 1 usage lyduybach2612
public void updateCareDiary(Long careDiaryId, CareDiaryDto careDiaryDto) {

    CareDiary careDiary = careDiaryRepository.findById(careDiaryId).get();
    CareDiaryMapper.updateCareDiaryFromCareDiaryDto(careDiaryDto, careDiary);
    careDiary.setId(careDiaryId);
    careDiaryRepository.save(careDiary);

}
```

- Với tính năng Update, hầu hết mọi thứ đều giống với tính năng Create. Tuy nhiên, trong URL tôi có thêm một PathVariable careDiaryId để có thể sửa dữ liệu với Id tương ứng. Bạn có thể tham khảo ở trên với tính năng Create.

### 5.7.4. Delete

Với chức năng cuối cùng trong CRUD, tôi tiến hành tạo ra một phương thức get phục vụ cho Delete. Tuy nhiên trong CareDiary tôi không tạo chức năng Delete nên tôi sẽ demo với một thực thể khác. Dưới đây là code demo.

```
@GetMapping("/delete/{productId}")  lyduybach2612
public String deleteProduct(@PathVariable Long productId) {

    productService.deleteProduct(productId);
    return "redirect:/products";

}
```

```
@Override  1 usage  lyduybach2612
public void deleteProduct(Long productId) {

    productRepository.deleteById(productId);

}
```

- Đây là demo cho chức năng xoá sản phẩm, trong Controller tôi gọi tới service để kết nối tới Repository và xoá dữ liệu trong cơ sở dữ liệu. Trong Service, tôi gọi tới Repository và xoá dữ liệu với id tương ứng.

## 5.8. Security Configuration

- Sau khi tạo xong các chức năng CRUD, tôi tiến hành cấu hình security để phục vụ cho việc Authentication và Authorization.

### 5.8.1. UserDetailService

- Đầu tiên, chúng ta tiến hành cấu hình UserDetailService phục vụ cho việc phân quyền.

```
@Service  lyduybach2612
@RequiredArgsConstructor
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
public class CustomUserServiceDetail implements UserDetailsService {

    ManagerRepository managerRepository;
    CustomerRepository customerRepository;
```

- Tôi tiến hành tạo ra class CustomUserServiceDetail implement lại UserDetailService. Tại đây tôi tiến hành Inject ManagerRepository và CustomerRepository.

```
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

    Manager manager = managerRepository.findByManagerName(username);
    if (manager != null) {
        return new User(
                manager.getManagerName(),
                manager.getPassword(),
                createAuthorities( role: "ROLE_ADMIN")
        );
    }

    Customer customer = customerRepository.findByUsername(username);
    if (customer != null) {
        return new User(
                customer.getUsername(),
                customer.getPassword(),
                createAuthorities( role: "ROLE_CUSTOMER")
        );
    }
```

- Sau khi tạo class, tôi tiến hành Override lại phương thức loadUserByUsername, tại đây tôi sẽ lấy user từ manager

và customer thông qua repository, sau đó set username, password và Role.

- Trong phương thức trên tôi có gọi tới phương thức createAuthorities là phương thức tôi tạo ở cùng class nhằm phục vụ mục đích tạo ra Role tương ứng với các vai trò.

```
private List<SimpleGrantedAuthority> createAuthorities(String role) {  2 usages  lyduybach2612
    return Collections.singletonList(new SimpleGrantedAuthority(role));
}
```

### 5.8.2. Security Configuration

- Đầu tiên, tôi sử dụng 2 annotation @Configuration và @EnableWebSecurity để thông báo cho ứng dụng biết đây là một class nhằm phục vụ cho việc config.

```
12
13  @Configuration  lyduybach2612
14  @EnableWebSecurity
15  public class SecurityConfig {
```

- Sau khi khai báo, tôi tiến hành config SecurityFilterChain. Mỗi một request khi được gửi tới Server sẽ phải đi qua các SecurityFilter, nhiều Filter sẽ tạo thành 1 Chain. Các Filter sẽ tiến hành kiểm tra về authentication, authorization, session,…

```
@Bean  lyduybach2612
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

    http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(authorize -> authorize
                    .requestMatchers(MANAGER_PERMIT).hasRole("ADMIN")
                    .requestMatchers(CUSTOMER_PERMIT).hasRole("CUSTOMER")
                    .requestMatchers(PERMIT_ALL).permitAll()
                    .anyRequest().authenticated()
            )
            .formLogin(form -> form
                    .loginPage("/login")
                    .defaultSuccessUrl("/products")
                    .loginProcessingUrl("/login")
                    .failureUrl( authenticationFailureUrl: "/login?fail")
                    .permitAll())
            .logout(logout->logout.logoutRequestMatcher(new AntPathRequestMatcher("/logout")));
    return http.build();
```

- Tôi sẽ khai báo phương thức này là một Bean để cho Spring IoC container quản lý. Trong cấu hình, tôi sẽ tắt csrf – một cách mà Spring cấu hình mặc định cho việc ngăn ngừa csrf vì không cần thiết. Sau đó tôi tiến hành cung cấp Authorization cho các Role.
- Sau khi cấu hình authorization xong, tôi tiến hành cấu hình các URL cho các chức năng đăng nhập, đăng kí.
- Sau khi cấu hình xong cho SecurityFilterChain, tôi tiến hành tạo một Bean passwordEncoder để Encode Password theo chuẩn BCrypt.

```
@Bean  lyduybach2612
public BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

- Cuối cùng tôi tạo một phương thức config nhằm trước khi lưu trữ password trong cơ sở dữ liệu tôi sẽ encode password trước.

```
public void configure(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception {  no usages  lyduybach2
    authenticationManagerBuilder.userDetailsService(customUserServiceDetail).passwordEncoder(passwordEncoder());
}
```

### 5.8.3. Session Config

- Tôi tiến hành tạo ra class SecurityUtil để lấy lại session người dùng.

```
public class SecurityUtil {  20 usages  lyduybach2612

    public static String getSessionUser(){  12 usages  lyduybach2612
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        if(!(authentication instanceof AnonymousAuthenticationToken)){
            return authentication.getName();
        }
        return null;
    }

}
```

- Trong đoạn code trên, tôi tạo ra một phương thức tĩnh (static) có tên getSessionUser để lấy lại thông tin Session của người dùng.

## 5.9. Chỉnh sửa lại các chức năng tương ứng với các vai trò sau khi cấu hình Security

Do việc chỉnh sửa sẽ tương đối giống nhau và thời gian không cho phép, tôi xin phép chỉ demo một chức năng trong CRUD. Dưới đây là demo cho chức năng tạo nhật kí chăm sóc khách hàng.

```
@RequiredArgsConstructor
public class CareDiaryController {

    CareDiaryService careDiaryService;
    ManagerService managerService;
    CustomerService customerService;
```

- Trước khi bắt đầu, tôi cần Inject các đối tượng cần thiết vào Controller.

```
    @GetMapping("/new")  lyduybach2612
    public String createCareDiaryForm(Model model) {

        String username = SecurityUtil.getSessionUser();

        Manager manager = managerService.findByUsername(username);
        boolean isManager = manager != null;
        model.addAttribute( attributeName: "isManager", isManager);
        CareDiaryDto careDiary = new CareDiaryDto();
        model.addAttribute( attributeName: "careDiary", careDiary);
        return "create-care-diary";

    }
```

- Như có thể thấy trên code, trước khi tạo một đối tượng thuộc lớp CareDiaryDto, tôi tiến hành lấy Session người dùng. Sau đó, tôi sẽ gọi tới đối tượng của ManagerService để tiến hành kiểm tra người dùng có tồn tại không. Tôi tạo ra một biến boolean để kiểm tra xem người dùng có phải manager không, sau đó sẽ trả biến đó về View.

```java
@Override  1 usage  lyduybach2612
public void createCareDiary(CareDiaryDto careDiaryDto) {

    String username = SecurityUtil.getSessionUser();
    Customer customer = customerRepository.findByUsername(username);
    careDiaryDto.setState(false);
    CareDiary careDiary = CareDiaryMapper.mapToCareDiary(careDiaryDto);
    careDiary.setCustomer(customer);
    careDiaryRepository.save(careDiary);

}
```

- Trong phương thức post của controller, sẽ không có gì thay đổi. Tuy nhiên, trong Service chúng ta sẽ có một số cập nhập. Cụ thể, trước khi lưu người dùng vào cơ sở dữ liệu, tôi tiến hành lấy Session người dùng và thông qua CustomerRepository, tôi set customer của nhật kí khách hàng là người dùng của Session đó.
- Trên đây là toàn bộ thay đổi của một trong những chức năng CRUD. Các chức năng còn lại, các bạn hoàn toàn có thể tham khảo và thực thi dựa theo nguyên lý trên.

6. Nhược điểm

Do một số yếu tố như thời gian và phát triển theo hướng cá nhân nên dự án vẫn còn tồn đọng một số nhược điểm như: Giao diện chưa được thiết kế tốt nhất, các chức năng vẫn có thể mở rộng thêm, tối ưu bảo mật với các kĩ thuật tốt hơn,… Những nhược điểm này có thể làm ảnh hưởng tới trải nghiệm của khách hàng. Tôi sẽ cố gắng hoàn thiện sản phẩm theo hướng phát triển và tích hợp sử dụng các công nghệ mới hơn.

7. Kết luận

Sau khoảng một tuần triển khai, dự án Smart Sales đã được hoàn thành với mục tiêu là một sản phẩm hỗ trợ khách hàng có thể dễ dàng đặt mua các sản phẩm cũng như gửi các yêu cầu hỗ

trợ để được chăm sóc khi cần thiết. Với các mẫu kiến trúc MVC, 3 – layer và DTO, dự án không chỉ đảm bảo tính bảo mật, ổn định mà còn dễ dàng bảo trì và mở rộng. Các tính năng cơ bản, chủ yếu được giới thiệu ở trên đã được tích hợp một cách hiệu quả, giúp cải thiện trải nghiệm của người dùng. Tuy nhiên bên cạnh đó vẫn còn tồn tại một số nhược điểm như chưa tối ưu giao diện. Trong tương lai, các cải tiến và mở rộng sẽ được tiếp tục và tập trung cải thiện vào những nhược điểm còn tồn đọng để có thể đáp ứng được nhu cầu của người dùng và thị trường.