Yolo Farm

Requirement

Functional Requirement

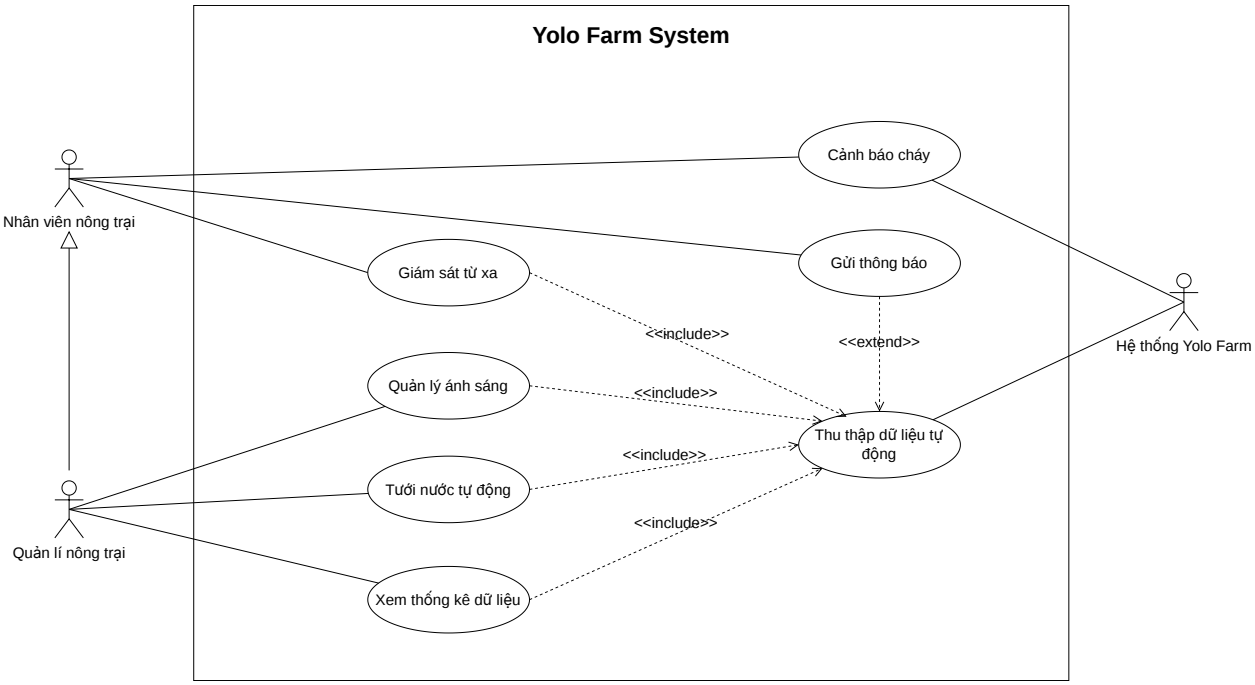
- Thu thập dữ liệu tự động
    - Hệ thống yolo-farm đo nhiệt độ và độ ẩm không khí bằng cảm biến nhiệt độ và độ ẩm không khí DHT 20
    - Hệ thống yolo-farm đo độ ẩm đất bằng cảm biến độ ẩm đất
    - Hệ thống yolo-farm đo cường độ ánh sáng bằng cảm biến ánh sáng
    - Hệ thống yolo-farm đo lượng khói để báo cháy bằng cảm biến MQ2
    - Hệ thống tự động gửi các thông tin đã đo được lên Cloud Server định kì
- Quản lý ánh sáng
    - Người quản lý có thể nhấn nút bật/tắt bóng đèn thông qua ứng dụng web app.
    - Người quản lý có thể sử dụng web app để điều chỉnh, thiết lập lịch bật / tắt bóng đèn tự động theo ngày, giờ. Đúng ngày giờ đã chọn, hệ thống sẽ tự động kích hoạt bóng đèn để cung cấp ánh sáng cho cây
    - Người quản lý có thể tùy chỉnh ánh sáng phù hợp cho cây trồng, khi cường độ ánh sáng đạt đến một ngưỡng nhất định gây hại cho cây thì hệ thống có thể tự động bật / tắt bóng đèn led dựa vào thông tin ánh sáng được thu thập từ Cloud Server
- Tưới nước tự động
    - Người quản lý có thể nhấn nút bật/tắt tưới nước thông qua ứng dụng web app.
    - Người quản lý có thể sử dụng web app để điều chỉnh, thiết lập lịch tưới nước tự động theo ngày, giờ. Đúng ngày giờ đã chọn, hệ thống sẽ tự động kích hoạt máy bơm nước hoạt động để tưới nước cho cây.

    - Người quản lý có thể tùy chỉnh ngưỡng độ ẩm phù hợp cho cây trồng, khi độ ẩm đạt đến một ngưỡng nhất định gây hại cho cây thì hệ thống có thể tự động kích hoạt hoặc ngắt hoạt động của máy bơm nước dựa vào thông tin độ ẩm được thu thập từ Cloud Server
- Thống kê dữ liệu
    - Hệ thống sẽ tự động lưu lại và thống kê các thông số (nhiệt độ, độ ẩm không khí, độ ẩm đất. cường độ ánh sáng) để hiển thị lên dashboard cho người dùng theo ngày, giờ cụ thể để người dùng có thể xem lại lịch sử các thông số trong khu vườn.
    - Cung cấp báo cáo tổng hợp về hiệu suất hoạt động của hệ thống, đề xuất tối ưu hóa nông trại dựa trên dữ liệu lịch sử (nếu có)
- Giám sát từ xa qua Web App
    - Lấy dữ liệu từ Cloud Server để hiển thị thông số mà cảm biến đã thu thập (nhiệt độ, độ ấm không khí, độ ẩm đất, ánh sáng) trên giao diện Web App
    - Hiển thị trạng thái của hệ thống (máy bơm nước, bóng đèn led, cảm biến,...)
- Gửi thông báo qua Web App
    - Thông báo dữ liệu bất thường qua giao diện (nhiệt độ, độ ấm không khí, độ ẩm đất, ánh sáng không nằm trong khoảng dữ liệu cho phép; dữ liệu bất thường không thể xử lí có thể do lỗi từ thiết bị phần cứng)
    - Thông báo tích cực từ hệ thống (tưới nước tự động thành công)
    - Thông báo kiểm tra nguồn nước tưới, vệ sinh kính nhà kính và bảo trì thiết bị định kỳ.
- Điều khiển thiết bị
    - Cho phép người dùng bật/tắt thủ công các thiết bị như tưới cây, quạt, đèn LED, chuông báo.
    - Nhận lệnh điều khiển từ remote hồng ngoại.
- Cảnh báo cháy
    - Hệ thống gửi thông báo cảnh báo cháy dựa vào thông tin từ cảm biến cháy nổ MQ2

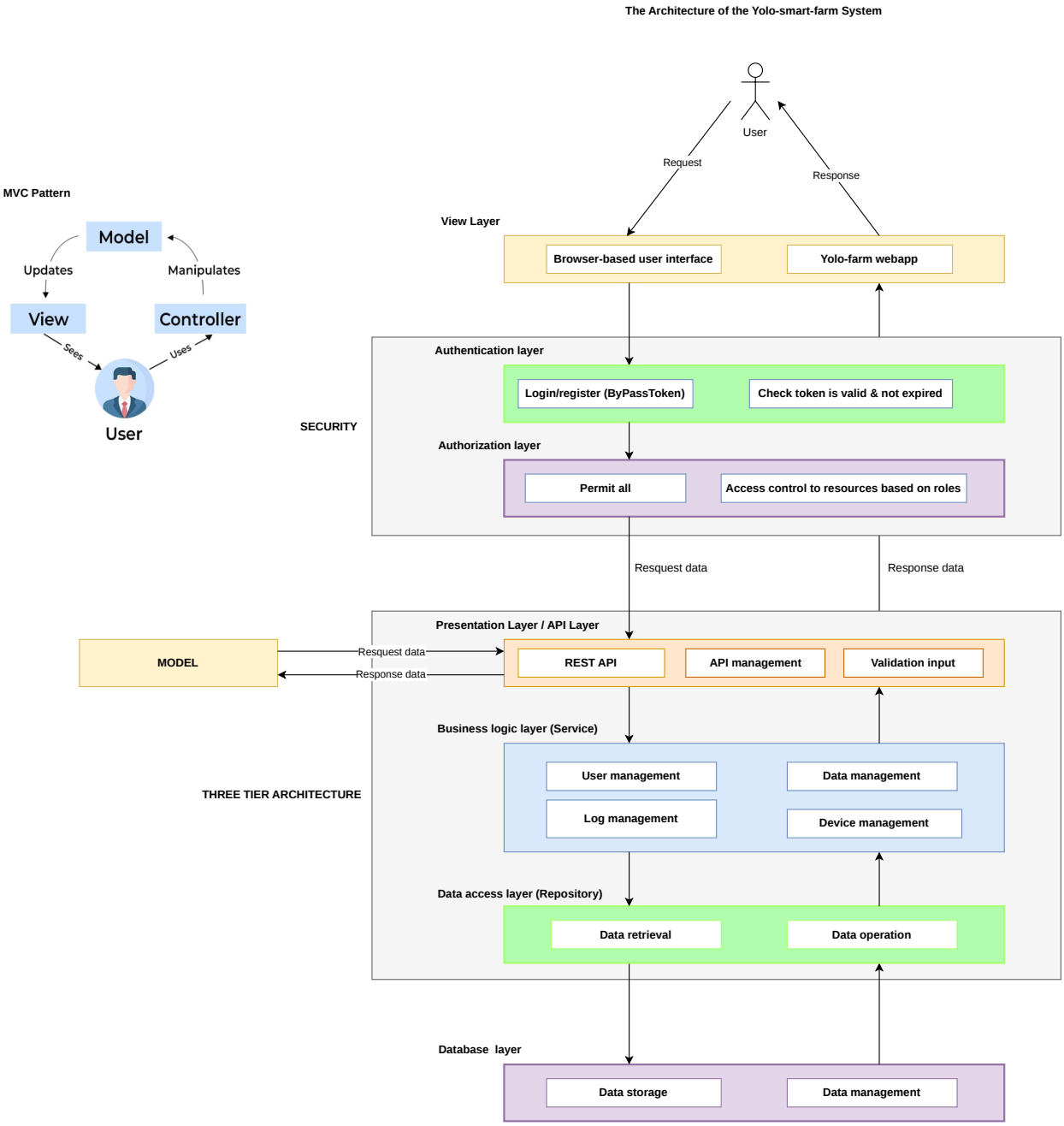
Non-Function Requirement

- Đảm bảo tính sẵn sàng và độ ổn định
  - Đảm bảo hệ thống không bị gián đoạn quá lâu, hoạt động ổn định và xuyên suốt 24/7.
  - Hỗ trợ dự phòng và khôi phục dữ liệu để đối phó với sự cố như thiên tai hoặc mất điện.
- Giao diện trực quan, thân thiện với người dùng
  - Giao diện người dùng cần phải thân tiện, UI/UX đơn giản và dễ sử dụng, hỗ trợ tiếng Anh/Việt.
  - Màu sắc tương phản tốt, dễ nhìn trong các điều kiện ánh sáng và thời tiết
  - Hỗ trợ iOS 14+, Android 10+, trình duyệt Chrome/Firefox mới nhất.
  - Tuân thủ GDPR và Luật pháp nhà nước Việt Nam.
- Hiệu suất cao và độ trễ thấp
  - Hiệu suất hệ thống cao, thời gian phản hồi cho người dùng nhanh chóng
  - Thông báo ngay lập tức đối với các chức năng đảm bảo an toàn như cảnh báo cháy.
- Đảm bảo bảo mật và tính tin cậy của hệ thống
  - Giới hạn số lần thực thi để tránh spam.
  - Phân quyền người dùng rõ ràng, tránh truy xuất vào các tài nguyên không được ủy quyền.
  - Sai số đối với mỗi cảm biến không được quá lớn so với thực tế
- Thân thiện với môi trường:
  - Trong quá trình triển khai hệ thống, cần tránh các hậu quả đối với môi trường và làm giảm năng suất cây trồng
- Dễ bảo trì và mở rộng:
  - Hệ thống cần hỗ trợ tích hợp và mở rộng các tính năng theo yêu cầu của người quản lý nông trại

- Dễ dàng mở rộng và nâng cấp để hỗ trợ các cảm biến mới, không gây lỗi cho hệ thống khi thêm cảm biến



Kiến trúc hệ thống



Mô hình kiến trúc phân lớp (Layered Architecture) mà nhóm thiết kế là sự kết hợp giữa kiến trúc 3 tầng (Three-tier architecture) và mẫu kiến trúc phần mềm MVC (MVC design pattern), kết hợp với 1 tầng bảo mật để cung cấp xác thực và phân quyền rõ ràng cho người dùng. Kiến trúc này bao gồm có 7 lớp chính: View Layer, Authentication Layer, Authorization Layer, Presentation Layer, Business Logic Layer, Data Access Layer và Database Layer. Trong đó, kiến trúc 3 tầng (Three-tier architecture) được thể hiện ở 3 lớp: Presentation Layer, Business Logic Layer

và Data Access Layer. Phần bảo mật (Security) của hệ thống được chia thành 2 lớp: Authentication Layer và Authorization Layer. Ngoài ra, MVC design pattern (Model - View - Controller) được sử dụng thông qua sự liên kết giữa các lớp: View Layer (View), Presentation Layer (Controller) và Model để tương tác (resquest - response) dữ liệu giữa các tầng của hệ thống.

Khi người dùng sử dụng hệ thống thì lớp thứ nhất là View Layer sẽ hiển thị giao diện để người dùng tương tác với hệ thống và hiển thị thông tin cho người dùng tại giao diện này. Thông qua giao diện trực quan, người dùng có thể dễ dàng thực hiện yêu cầu (request) đến hệ thống và có thể nhận dữ liệu trả ra (response) để đáp ứng yêu cầu người dùng.

Request của người dùng sẽ được nhận bởi lớp Presentation layer (hay API layer), lớp này có tích hợp Security và tầng Controller để bắt và xử lí RESTful API

Khi thực hiện một thao tác request đến hệ thống, người dùng phải đi qua phần bảo mật (Security) của hệ thống để đảm bảo xác thực và có thể truy cập được đúng loại tài nguyên mà mình được phần quyền, tránh người dùng có thể xem được những thông tin nhạy cảm ảnh hưởng đến tính bảo mật của hệ thống. Tầng security này chia thành 2 lớp:

- Authentication Layer: Lớp này thực hiện xác thực người dùng trước khi cho phép truy cập vào các tài nguyên của hệ thống. Việc xác thực này bao gồm đăng kí tài khoản và đăng nhập để sử dụng hệ thống. Khi đăng kí tài khoản thành công, hệ thống sẽ thực hiện thêm một người dùng vào cơ sở dữ liệu và lưu lại tài khoản, mật khẩu kèm với vai trò và một số thông tin khác của người dùng. Khi đăng nhập thành công (đúng tài khoản và mật khẩu), người dùng sẽ được cấp một token (JSON Web Token) tương ứng với người dùng đó, token sẽ được gửi kèm với các yêu cầu của người dùng ở trong phiên đăng nhập để xác thực yêu cầu một cách liên tục. Token cũng sẽ có thời hạn, khi quá hạn, người dùng cần đăng nhập lại để có thể tiếp tục thực hiện các yêu cầu.
- Authorization Layer: Lớp này sẽ thực hiện việc phân quyền tài nguyên hệ thống cho người dùng. Một người dùng với vai trò cụ thể (sinh viên hoặc SPSO) sẽ có quyền truy cập một số tài nguyên cụ thể mà không phải toàn bộ hệ thống để tránh có thể xem được thông tin nhạy cảm của hệ thống hoặc của người dùng khác, đó là lí do lớp này được sử dụng.

Sau khi qua được lớp bảo mật của hệ thống, Controller sẽ bắt các RESTful API và thực hiện thao tác với Model (trong MVC design pattern) để thực hiện các thao tác bên trong hệ thống. Lớp này sau khi thực hiện kiểm tra tính đúng đắn của dữ liệu đầu vào thông qua (validation) sẽ gọi đến các hàm ở tầng bên dưới để xử lí logic. Ở chiều ngược lại, khi nhận dữ liệu trả ra từ các tầng bên dưới thì sẽ thông qua Model (trong MVC design pattern) để trả trực tiếp ra View (không thông qua Security)

Business Logic Layer (Service) là lớp chịu trách nhiệm triển khai các chức năng cốt lõi và logic nghiệp vụ của hệ thống. Lớp này nhận yêu cầu từ lớp Presentation, xử lý nó và gửi yêu cầu tương ứng đến lớp Data Access. Ở chiều ngược lại, chiều trả dữ liệu về, lớp này cũng sẽ thực hiện việc xử lí và chuẩn hóa dữ liệu server trả ra theo định dạng client mong muốn và gửi lên cho lớp Presentation.

Data access layer (Repository) là lớp đảm nhiệm việc nhận yêu cầu từ lớp Business logic và gửi yêu cầu đến lớp Database để thực hiện các thao tác liên quan đến dữ liệu.

Database layer là lớp thấp nhất trong kiến trúc phân lớp. Đây là lớp thực hiện việc lưu trữ, quản lý và thao tác dữ liệu theo cách được định nghĩa từ lớp Data Access. Ở đây, database mà nhóm sử dụng là MongoDB

Design Pattern

Trong quá trình xây dựng hệ thống, nhóm đã áp dụng nhiều mẫu thiết kế phần mềm (Design Pattern) nhằm đảm bảo cấu trúc hệ thống rõ ràng, dễ bảo trì, dễ kiểm thử và có khả năng mở rộng về sau. Các mẫu thiết kế này không được áp dụng rời rạc mà được tích hợp một cách chặt chẽ trong từng thành phần của hệ thống, bám sát vào mô hình kiến trúc phân lớp và tuân thủ các nguyên lý SOLID trong thiết kế phần mềm hiện đại. Dưới đây là phân tích chi tiết các Design Pattern được áp dụng trong hệ thống:

---

1. Dependency Injection Pattern (DI Pattern)

Dependency Injection là một trong những mẫu thiết kế cốt lõi được áp dụng trong hệ thống dựa trên nền tảng Spring Boot. Mẫu này cho phép các thành phần phụ thuộc được tự động tiêm vào các lớp thông qua các annotation như @Autowired,

giúp loại bỏ nhu cầu khởi tạo thủ công và giảm sự phụ thuộc chặt chẽ giữa các thành phần.

Ví dụ mã nguồn sử dụng trong hệ thống:

```
@Autowired
private IDataService dataService;
```

Trong đoạn code trên, `DataController` không cần khởi tạo `DataService` một cách thủ công mà Spring sẽ tự động cung cấp một instance phù hợp tại thời điểm runtime. Cách tiếp cận này được áp dụng nhất quán trong tất cả các controller (`UserController`, `DataController`, ...) và service (`UserService`, `AuthService`, `DataService`, ...).

Cách tiếp cận này không chỉ giảm thiểu coupling giữa DataController và DataService mà còn hỗ trợ kiểm thử đơn vị (unit testing) bằng cách dễ dàng thay thế với các mock object.

---

2. Interface-based Programming (Dependency Inversion Principle)

Hệ thống tuân thủ nguyên lý "Lập trình hướng interface", trong đó mọi class nghiệp vụ chính đều triển khai từ một interface tương ứng. Việc này giúp các thành phần không phụ thuộc trực tiếp vào nhau mà chỉ phụ thuộc vào abstraction (sự trừu tượng).

Ví dụ mã nguồn sử dụng trong hệ thống:

```
public interface IDataService {
    Map<String, Object> getMode(String userId, String factorName);
    ...
}

@Service
public class DataService implements IDataService { ... }
```

Trong hệ thống, `IDataService`, `IUserService`, `IAuthService`, ... đóng vai trò định nghĩa contract, trong khi `DataService`, `UserService`, ... thực hiện cụ thể logic.

Phương pháp này tăng cường tính linh hoạt, cho phép thay thế hoặc mở rộng chức năng mà không ảnh hưởng đến các thành phần khác. Ví dụ, trong quá trình kiểm thử, các interface như `IUserService` hay `IDataService` có thể được mock để mô phỏng hành vi mà không cần truy cập thực tế vào cơ sở dữ liệu. Ngoài ra, nếu cần sửa đổi logic hệ thống, có thể định nghĩa thêm class `DataService2` triển khai interface `IDataService` mà không cần thiết phải sửa đổi mã nguồn ở `DataService`

---

3. Repository Pattern

Repository Pattern được sử dụng để tách biệt logic nghiệp vụ khỏi logic truy cập dữ liệu, đảm bảo tính nhất quán và tái sử dụng trong việc quản lý dữ liệu. Trong hệ thống, mẫu này được triển khai thông qua các interface như `FactorRepository` và `DeviceRepository`,... kế thừa từ `MongoRepository` của Spring Data.

Ví dụ mã nguồn sử dụng trong hệ thống:

```
public interface FactorRepository extends MongoRepository<Factor, String> {
    Optional<Factor> findByUserIDAndName(String userID, String name);
}
```

Các repository trong hệ thống như `UserRepository`, `DeviceRepository`, `StatRepository`, ... đóng vai trò là trung gian giữa tầng Service và cơ sở dữ liệu MongoDB.

Mẫu thiết kế này giúp chuẩn hóa cách tiếp cận dữ liệu, giảm thiểu lỗi do mã hóa thủ công và hỗ trợ mở rộng khi cần tích hợp thêm các nguồn dữ liệu khác trong tương lai, chẳng hạn như chuyển đổi từ MongoDB sang cơ sở dữ liệu quan hệ.

---

4. Singleton Pattern

Các lớp được đánh dấu bởi annotation `@Service`, `@Repository`, hoặc `@Component` trong Spring Boot tự động được quản lý theo phạm vi Singleton, đảm bảo chỉ khởi tạo một instance duy nhất tồn tại trong suốt vòng đời ứng dụng.

Ví dụ mã nguồn sử dụng trong hệ thống:

```
@Service
public class UserService implements IUserService { ... }
```

`UserService` được khai báo với `@Service`, đảm bảo rằng mọi yêu cầu đều sử dụng cùng một instance

Việc đảm bảo singleton cho các service giúp tối ưu hóa tài nguyên hệ thống, đồng thời đảm bảo tính nhất quán (consistancy) trong các hoạt động nghiệp vụ.

---

5. Strategy Pattern (ứng dụng qua ánh xạ cấu hình)

Strategy Pattern được thể hiện trong `DataService`, xử lý linh hoạt các yếu tố môi trường như sensor, đơn vị đo lường, thiết bị điều khiển,... thông qua cấu trúc dữ liệu `Map` cấu hình tĩnh (FACTOR_DEVICE_MAP, FACTOR_FEED_MAP). Thay vì sử dụng các câu lệnh rẽ nhánh cứng, hệ thống tra cứu động bằng các `Map` cấu hình sẵn để ánh xạ yếu tố với thiết bị tương ứng.

Ví dụ mã nguồn sử dụng trong hệ thống:

```
private static final Map<String, String> FACTOR_DEVICE_MAP = Map.of(
    "moisture", "pump",
    "temperature", "fan",
    "light", "light",
    "humidity", "fan2"
);
```

Mỗi khi cần thao tác thiết bị theo yếu tố môi trường, hệ thống chỉ cần tra cứu `Map` thay vì dùng câu lệnh rẽ nhánh cứng. Cách tiếp cận này thể hiện tinh thần của Strategy Pattern, cho phép dễ dàng thêm hoặc sửa đổi các chiến lược xử lý mà không cần thay đổi mã nguồn chính, thể hiện rõ tinh thần tách biệt thuật toán khỏi luồng xử lý.

---

6. Template Method Pattern

Trong tất cả controller như `DataController`, `UserController`, các method xử lý REST API đều có cấu trúc lặp lại: xác thực người dùng → gọi service xử lý → trả về response → xử lý lỗi nếu có. Cấu trúc chung của các phương thức được định nghĩa như một khung mẫu, với các chi tiết cụ thể được triển khai bởi các service.

Ví dụ mã nguồn sử dụng trong hệ thống:

```
@GetMapping("/{factor}/mode")
public ResponseEntity<?> getMode(@PathVariable String factor) {
    try {
        return ResponseEntity.ok(dataService.getMode(getUserId(), factor));
    } catch (Exception e) {
        return ResponseEntity.status(404).body(Map.of("error", e.getMessage
()));
    }
}
```

Đây là minh họa rõ ràng cho Template Method Pattern, nơi xương sống của phương thức được xác định sẵn, trong khi các xử lý chi tiết được giao cho các hàm bên trong (ở đây là các service), đảm bảo tính nhất quán trong cách xử lý yêu cầu, đồng thời cho phép tùy chỉnh logic cụ thể thông qua các service bên dưới, tăng tính tái sử dụng và dễ bảo trì.

---

7. Observer Pattern (ứng dụng qua @Scheduled)

Observer Pattern được triển khai thông qua cơ chế `@Scheduled` trong `DataService`, cho phép hệ thống tự động quan sát và phản ứng với dữ liệu cảm biến theo chu kỳ. Phương thức `refreshAllFactors` được kích hoạt định kỳ để cập nhật trạng thái từ các cảm biến và kích hoạt thiết bị khi cần.

Ví dụ mã nguồn sử dụng trong hệ thống: `@Scheduled(fixedRate = 60000)` thực hiện kiểm tra và xử lý dữ liệu mỗi phút:

```
@Scheduled(fixedRate = 60000)
public void refreshAllFactors() {
    ...
}
```

Mỗi lần thực hiện, hệ thống sẽ kiểm tra dữ liệu thực tế từ sensor, ghi log và bật/tắt thiết bị tương ứng nếu vượt ngưỡng, cơ chế này mô phỏng hành vi quan sát liên tục, đảm bảo hệ thống phản ứng tự động với các thay đổi mà không cần sự can thiệp thủ công từ người dùng.