Domain-Driven Design trong Hospital Managment System

Giới thiệu ngắn

DDD là cách thiết kế hướng nghiệp vụ: code nên phản ánh trực tiếp các khái niệm nghiệp vụ (Patients, Appointments, Payments...). Mục tiêu ở HMS: **giảm độ phức tạp**, dễ mở rộng (thêm Billing/Notification/Imaging), tách biệt business rules khỏi infra, dễ test.

Kiến trúc tổng quan (4 layer)

HospitalManagementSystem.API/

- **Gồm những gì:**

Controllers/: Chứa các controller nhận request từ client (REST/gRPC).

Services/: Các adapter service cho external API hoặc gRPC endpoints.

Protos/: Định nghĩa contract gRPC để giao tiếp giữa các service.

Các file cấu hình như Program.cs, appsettings.json, Dockerfile.

- **Tại sao sử dụng tầng này:**

Đây là cổng vào của hệ thống, đảm bảo tất cả giao tiếp từ bên ngoài (client, ứng dụng mobile, hệ thống khác) đều qua một lớp rõ ràng. Nó tách biệt client khỏi logic nghiệp vụ thực sự.

- **Ý nghĩa:**

Đảm bảo separation of concerns: chỉ xử lý request/response, không chứa business logic.

Dễ dàng thay đổi giao diện giao tiếp (REST → gRPC) mà không ảnh hưởng business.

Giữ cho API luôn gọn nhẹ, chỉ làm nhiệm vụ chuyển dữ liệu vào Application Layer.

HospitalManagementSystem.Application/

- **Gồm những gì:**

Services/: Các orchestrator/use-case handler như NotificationServiceManager, JwtTokenService.

Chứa Commands, Queries, Validation logic, Transaction boundaries, publish event ra MQ khi cần.

- **Tại sao sử dụng tầng này:**

Đây là tầng điều phối luồng nghiệp vụ. Controller không nên chứa logic nghiệp vụ phức tạp, nên Application Layer đứng giữa để xử lý orchestration, giao tiếp với Domain.

- **Ý nghĩa:**

Giúp điều phối use-case nhiều bước (ví dụ: đặt lịch hẹn → gửi notification → cập nhật cache).

Đảm bảo transaction và xử lý cross-cutting (validation, exception handling).

Dễ test: Application layer có thể test độc lập bằng cách mock Domain/Infrastructure.

Cung cấp một lớp "trung gian" linh hoạt, giúp hệ thống dễ thay đổi hoặc thêm mới use-case.

HospitalManagementSystem.Domain/

- **Gồm những gì:**

Entities/: Các entity cốt lõi (Patient, Doctor, Appointment, Billing, User...).

Events/: Các domain event (AppointmentCreatedEvent, PaymentProcessedEvent...).

Repositories/: Interface repository (IappointmentRepository, IbillingRepository...).

Factories/: Interface factory (IpaymentFactory).

Payments/, Notifications/, Storages/, Caching/: Interface nghiệp vụ.

RabbitMQ/: Interface publish/consume event.

- **Tại sao sử dụng tầng này:**

Đây là trái tim nghiệp vụ (business core), tập trung toàn bộ khái niệm cốt lõi và quy tắc của HMS. Nó phải độc lập, không phụ thuộc vào công nghệ (EF, Redis, MQ...).

- **Ý nghĩa:**

Định nghĩa domain model phản ánh trực tiếp nghiệp vụ thực tế (patients, appointments, billing...).

Tách biệt business rules khỏi hạ tầng, giúp dễ thay đổi công nghệ mà không ảnh hưởng logic.

Tính mở rộng cao: chỉ cần thêm implement mới ở Infrastructure, Domain không đổi.

Là nền tảng để đảm bảo hệ thống đúng nghiệp vụ và dễ test.

HospitalManagementSystem.Infrastructure/

- **Gồm những gì:**

Persistence/: DbContext, migrations (EF Core).

Repositories/: Implement repository (AppointmentRepository, BillingRepository...).

Caching/: RedisCacheService, CachedPatientRepository.

Channels/: EmailNotificationChannel, SmsNotificationChannel, PushNotificationChannel.

PaymentMethods/: StripePaymentMethod, CashPaymentMethod.

PaymentFactory/: Factory tạo payment method.

RabbitMQ/: RabbitMQService, RabbitMQConsumerService.

Storage/: SeaweedStorageService.

BillingStrategies/: InsuranceBillingStrategy, BillingStrategyFactory.

- **Tại sao sử dụng tầng này:**

Đây là tầng kết nối với thế giới bên ngoài (database, message broker, external payment API). Toàn bộ implement của interface ở Domain nằm tại đây.

- **Ý nghĩa:**

Cho phép thay đổi công nghệ (SQL → NoSQL, RabbitMQ → Kafka, Redis → In-memory) mà không ảnh hưởng Domain/Application.

Đảm bảo DI (Dependency Injection): Application chỉ cần interface, Infrastructure cung cấp implement.

Giữ code kỹ thuật ở một nơi, tránh lẫn vào business logic.

Hỗ trợ event-driven architecture (RabbitMQ, event publish/consume).

Domain

Vị trí trong kiến trúc:

- Domain Layer là trái tim nghiệp vụ của hệ thống.
- Nằm ở trung tâm kiến trúc DDD, phía dưới Application Layer và phía trên Infrastructure Layer.

Entities/

- Chứa: Các class đại diện cho đối tượng nghiệp vụ cốt lõi của hệ thống.
- Ví dụ: Patient, Doctor, Appointment, Billing, Notification, User, RefreshToken, ...
- Ý nghĩa: Định nghĩa thuộc tính, hành vi, quy tắc nghiệp vụ của từng đối tượng. Là trung tâm của Domain, mọi logic nghiệp vụ đều xoay quanh các entity này.

Repositories/

- Chứa: Các interface định nghĩa cách truy xuất, lưu trữ entity từ/to database hoặc các nguồn dữ liệu khác.

- Ví dụ: IAppointmentRepository, IBillingRepository, IDoctorRepository, ...
- Ý nghĩa: Tách biệt logic nghiệp vụ khỏi logic lưu trữ. Giúp dễ dàng thay đổi cách lưu trữ (SQL, NoSQL, In-memory, ...), chỉ cần thay implement ở Infrastructure.

Factories/

- Chứa: Các interface định nghĩa cách tạo ra các đối tượng nghiệp vụ phức tạp.
- Ví dụ: IPaymentFactory.
- Ý nghĩa: Đảm bảo việc tạo đối tượng đúng quy tắc nghiệp vụ, có thể thay đổi cách tạo mà không ảnh hưởng nơi sử dụng.

Payments/

- Chứa: Các interface định nghĩa các phương thức thanh toán.
- Ví dụ: IPaymentMethod.
- Ý nghĩa: Cho phép hệ thống mở rộng nhiều loại thanh toán (Stripe, Cash, Insurance, ...). Tách biệt logic nghiệp vụ khỏi logic gọi API thanh toán.

Notifications/

- Chứa: Interface định nghĩa các kênh gửi thông báo.
- Ví dụ:
- Ý nghĩa: Hỗ trợ gửi thông báo qua nhiều kênh (Email, SMS, Push). Dễ dàng mở rộng thêm kênh mới mà không ảnh hưởng nghiệp vụ.

RabbitMQ/

- Chứa: Interface định nghĩa cách publish/consume event nghiệp vụ qua message broker.
- Ví dụ: IRabbitMQService.
- Ý nghĩa: Tách biệt logic nghiệp vụ khỏi logic kỹ thuật MQ. Dễ dàng thay đổi MQ (RabbitMQ, Kafka, ...) mà không ảnh hưởng Domain.

Storages/

- Chứa: Interface định nghĩa cách lưu trữ file, hình ảnh, tài liệu.

- Ví dụ: IStorageService.
- Ý nghĩa: Hỗ trợ lưu trữ nhiều loại (local, cloud, SeaweedFS, ...). Tách biệt nghiệp vụ khỏi logic lưu trữ kỹ thuật.

Caching/

- Chứa: Interface định nghĩa cách cache dữ liệu nghiệp vụ.
- Ví dụ: ICacheService.
- Ý nghĩa: Tăng hiệu năng truy xuất dữ liệu. Dễ dàng thay đổi cache (Redis, Memory, ...) mà không ảnh hưởng nghiệp vụ.

Events/

- Chứa: Các class đại diện cho sự kiện nghiệp vụ (Domain Event) đã xảy ra trong hệ thống.
- Ví dụ: AppointmentCreatedEvent, AppointmentCancelledEvent, PaymentProcessedEvent, RefundProcessedEvent, IDomainEvent.
- Ý nghĩa: Ghi nhận các mốc nghiệp vụ quan trọng (ví dụ: lịch hẹn được tạo, thanh toán hoàn tất). Dùng để publish lên MQ, trigger các hành động khác (notification, audit, integration). IDomainEvent là interface chung cho mọi event, giúp event bất biến và có thời điểm xảy ra.

Ví dụ cụ thể về AppointmentCreatedEvent trong Hospital Management System:

- Định nghĩa Event: AppointmentCreatedEvent
    - Event này là một Domain Event, đại diện cho sự kiện nghiệp vụ: "Một cuộc hẹn mới đã được tạo".

```
namespace HospitalManagementSystem.Domain.Events
{
    public class AppointmentCreatedEvent : IDomainEvent
    {
        public int AppointmentId { get; }
        public int PatientId { get; }
        public string PatientName { get; }
        public int DoctorId { get; }
        public string DoctorName { get; }
```

```
        public string DoctorSpecialty { get; }
        public DateTime Date { get; }
        public string Status { get; }
        public DateTime CreatedAt { get; }
        public int CreatedByUserId { get; }
        public string CreatedByRole { get; }
        public DateTime OccurredOn { get; }

        public AppointmentCreatedEvent(
            int appointmentId,
            int patientId,
            string patientName,
            int doctorId,
            string doctorName,
            string doctorSpecialty,
            DateTime date,
            string status,
            DateTime createdAt,
            int createdByUserId,
            string createdByRole)
        {
            AppointmentId = appointmentId;
            PatientId = patientId;
            PatientName = patientName;
            DoctorId = doctorId;
            DoctorName = doctorName;
            DoctorSpecialty = doctorSpecialty;
            Date = date;
            Status = status;
            CreatedAt = createdAt;
            CreatedByUserId = createdByUserId;
            CreatedByRole = createdByRole;
            OccurredOn = DateTime.UtcNow;
        }
    }
}
```

- Đặc điểm:

    - Bất biến (chỉ có getter, set qua constructor).
    - Có đầy đủ thông tin nghiệp vụ liên quan đến sự kiện.
    - Có OccurredOn để ghi nhận thời điểm sự kiện xảy ra.
- Luồng đi của Event trong hệ thống
    - Tạo Event ở Application/API Layer: Khi một cuộc hẹn mới được tạo qua API, controller sẽ khởi tạo event này:

```
await _rabbitMQService.PublishAppointmentCreatedAsync(
    new AppointmentCreatedEvent(
        createdAppointment.Id,
        createdAppointment.PatientId,
        patient.Name,
        createdAppointment.DoctorId,
        doctor.Name,
        doctor.Specialty,
        createdAppointment.Date,
        createdAppointment.Status,
        createdAppointment.CreatedAt,
        currentUserId,
        currentUserRole
    )
);
```

- Publish Event ở Infrastructure Layer: RabbitMQService nhận event, serialize và publish lên RabbitMQ:

```
public async Task PublishAppointmentCreatedAsync(AppointmentCreated
Event appointmentData)
            ⇒ await PublishEventAsync(_hospitalExchange, "appointment.creat
ed", appointmentData);
```

- Consume Event ở Infrastructure Layer: RabbitMQConsumerService lắng nghe queue, nhận event, deserialize và xử lý:

```
object eventData = routingKey switch
{
```

```
        "appointment.created" => JsonConvert.DeserializeObject<AppointmentCreatedEvent>(message!),
        // ...
};
if (eventData != null)
{
        Notification notification = CreateAppointmentCreatedNotification((AppointmentCreatedEvent)eventData, patientRepository);
        // Lưu notification vào DB, gửi email qua EmailNotificationChannel
}
```

- Xử lý hậu sự kiện:
  - Gửi email xác nhận cho bệnh nhân.
  - Lưu notification vào database.
  - Có thể trigger các hành động khác: gửi SMS, push notification, audit log, tích hợp với các service khác.

Application

Vị trí trong kiến trúc:

- Nằm giữa Presentation (API) và Domain.
- Là nơi điều phối các use-case nghiệp vụ, xử lý logic luồng đi, transaction, validation cross-cutting, publish event.

Services/

- Chứa: Các class điều phối nghiệp vụ, orchestrator, use-case handler, manager.
- Ví dụ:
  - JwtTokenService: Điều phối việc tạo JWT cho user, gọi các interface xác thực ở Domain.
  - NotificationServiceManager: Điều phối gửi notification, gọi các kênh thông báo qua interface INotificationChannel.
  - PatientCacheWarmupService: Điều phối việc warmup cache cho bệnh nhân, gọi interface cache ở Domain.

- Ý nghĩa:
  - Tập trung xử lý các luồng nghiệp vụ phức tạp, nhiều bước.
  - Tách biệt logic orchestration khỏi controller (API) và khỏi logic kỹ thuật (Infrastructure).
  - Dễ mở rộng, dễ test, dễ bảo trì.

Luồng đi của Application Layer

- Controller (API Layer) nhận request → gọi service ở Application Layer.
- Service ở Application Layer:
  - Validate dữ liệu, kiểm tra quyền, bắt lỗi.
  - Gọi các repository, service, strategy, factory qua interface ở Domain Layer.
  - Có thể publish event nghiệp vụ (Domain Event) khi cần.
  - Điều phối các bước nghiệp vụ (ví dụ: tạo bệnh nhân → gửi thông báo → cập nhật cache).
  - Kết quả trả về cho Controller → trả response cho client.

Ví dụ cụ thể: NotificationServiceManager

```
public class NotificationServiceManager

{

    private readonly IEnumerable<INotificationChannel> _channels;

        public NotificationServiceManager(IEnumerable<INotificationChannel
> channels)

        {

            _channels = channels;

        }
```

```
        public async Task SendAsync(NotificationMessage message)
        {
            foreach (var channel in _channels)
        {
            await channel.SendAsync(message);
        }
    }
}
```

- Ý nghĩa:
  - Điều phối gửi notification qua nhiều kênh (email, sms, push).
  - Không biết chi tiết từng kênh gửi như thế nào (đã tách qua interface ở Domain và implement ở Infrastructure).
  - Dễ mở rộng thêm kênh mới chỉ bằng cách thêm implement ở Infrastructure.

Infrastructure

Vị trí trong kiến trúc:

- Infrastructure Layer là tầng thấp nhất trong kiến trúc DDD.
- Nằm dưới Domain và Application Layer, kết nối hệ thống với các công nghệ bên ngoài.

BillingStrategies/

- Chứa: Implement các thuật toán tính hóa đơn, bảo hiểm, ... (implement của IBillingStrategy, IBillingStrategyFactory).
- Ví dụ:
  - InsuranceBillingStrategy: Tính hóa đơn theo bảo hiểm.

    - BillingStrategyFactory: Chọn strategy phù hợp.
- Ý nghĩa: Cho phép mở rộng nhiều cách tính hóa đơn mà không ảnh hưởng Domain.

Caching/

- Chứa: Implement các giải pháp cache (implement của ICacheService).
- Ví dụ:
    - RedisCacheService: Lưu cache bằng Redis.
    - CachedPatientRepository: Decorator cache cho repository.
    - CacheKeyGenerator: Sinh key cache.
- Ý nghĩa: Tăng hiệu năng truy xuất dữ liệu, dễ thay đổi giải pháp cache.

Channels/

- Chứa: Implement các kênh gửi thông báo (implement của INotificationChannel).
- Ví dụ:
    - EmailNotificationChannel: Gửi email.
    - SmsNotificationChannel: Gửi SMS.
    - PushNotificationChannel: Gửi push notification.
- Ý nghĩa: Dễ dàng mở rộng thêm kênh mới, chỉ cần thêm implement.

PaymentFactory/

- Chứa: Implement factory tạo payment method (implement của IPaymentFactory).
- Ví dụ: PaymentFactory: Tạo đối tượng payment method phù hợp.
- Ý nghĩa: Dễ mở rộng thêm phương thức thanh toán mới.

PaymentMethods/

- Chứa: Implement các phương thức thanh toán (implement của IPaymentMethod).
- Ví dụ:

    - StripePaymentMethod: Thanh toán qua Stripe.
    - CashPaymentMethod: Thanh toán tiền mặt.
- Ý nghĩa: Tách biệt logic gọi API thanh toán khỏi nghiệp vụ.

Persistence/

- Chứa: DbContext, cấu hình EF Core, migration.
- Ví dụ:
    - HospitalDbContext: Quản lý truy xuất database.
    - Ý nghĩa: Kết nối, quản lý database, mapping entity.

RabbitMQ/

- Chứa: Implement publish/consume event qua message broker (implement của IRabbitMQService).
- Ví dụ:
    - RabbitMQService: Publish event lên RabbitMQ.
    - RabbitMQConsumerService: Consume event từ RabbitMQ.
- Ý nghĩa: Tích hợp hệ thống event-driven, dễ thay đổi MQ.

Repositories/

- Chứa: Implement các repository (implement của IAppointmentRepository, IBillingRepository, ...).
- Ví dụ: AppointmentRepository, BillingRepository, PatientRepository, ...
- Ý nghĩa: Thực hiện truy xuất, lưu trữ entity vào database.

Storage/

- Chứa: Implement lưu trữ file, hình ảnh (implement của IStorageService).
- Ví dụ: SeaweedStorageService: Lưu trữ file qua SeaweedFS.
- Ý nghĩa: Dễ dàng thay đổi giải pháp lưu trữ file.

Luồng đi của Infrastructure Layer

- Application/Domain gọi interface (repository, service, strategy, ...).
- Infrastructure cung cấp implement cho các interface đó qua DI.

- Thực hiện các thao tác kỹ thuật: truy xuất DB, lưu cache, gửi email, publish event, ...
- Kết quả trả về cho Application/Domain.

Ví dụ

- Định nghĩa interface cho phương thức thanh toán (IPaymentMethod) và factory (IPaymentFactory):

```
// HospitalManagementSystem.Domain/Payments/IPaymentMethod.cs

public interface IPaymentMethod

{

    string PaymentMethodName { get; }

    Task<PaymentResult> ProcessPaymentAsync(PaymentRequest request);

    Task<PaymentResult> RefundPaymentAsync(string transactionId, decim
al amount);

}
```

```
// HospitalManagementSystem.Domain/Factories/IPaymentFactory.cs

public interface IPaymentFactory

{

    IPaymentMethod CreatePaymentMethod(string paymentMethodName);

    IEnumerable<string> GetAvailablePaymentMethods();

}
```

- Infrastructure Layer: Implement các phương thức thanh toán cụ thể (Stripe, Cash):

```
//HospitalManagementSystem.Infrastructure/PaymentMethods/StripePaymentMethod.cs

public class StripePaymentMethod : IPaymentMethod

{

    public string PaymentMethodName => "Stripe";

    // ... constructor DI ...

    public async Task<PaymentResult> ProcessPaymentAsync(PaymentRequest request)

    {

        // Tạo Stripe Checkout Session, trả về checkout_url cho client

        // (Xem code chi tiết trong file bạn đã có)

    }

    public async Task<PaymentResult> RefundPaymentAsync(string transactionId, decimal amount)

    {

        // Gọi Stripe API để refund

    }

}
```

```
//HospitalManagementSystem.Infrastructure/PaymentMethods/CashPaymentMethod.cs

public class CashPaymentMethod : IPaymentMethod
```

```
{

    public string PaymentMethodName ⇒ "Cash";

    // ... constructor DI ...

    public async Task<PaymentResult> ProcessPaymentAsync(PaymentReq
uest request)

    {

        // Xử lý logic thanh toán tiền mặt (giả lập thành công)

    }

    public async Task<PaymentResult> RefundPaymentAsync(string transac
tionId, decimal amount)

    {

        // Xử lý logic hoàn tiền mặt (giả lập thành công)

    }

}
```

- Factory tạo đúng phương thức thanh toán:

```
// HospitalManagementSystem.Infrastructure/PaymentFactory/PaymentFac
tory.cs

public IPaymentMethod CreatePaymentMethod(string paymentMethodNam
e)

{

    // Trả về StripePaymentMethod hoặc CashPaymentMethod từ DI contain
```

```
er

}
```

- Sau khi ra khỏi Infrastructure
  - Kết quả trả về:
    - PaymentResult chứa thông tin thành công/thất bại, transactionId, checkout_url (Stripe), v.v.
    - Xử lý tiếp theo ở Application/API:
      - Nếu là Stripe, trả về checkout_url cho client để redirect sang Stripe.
      - Nếu là Cash, cập nhật billing status thành "Completed" ngay.
      - Publish event lên RabbitMQ (PaymentProcessedEvent, PaymentInitiatedEvent, PaymentFailedEvent).

API

Vị trí trong kiến trúc:

- Là cổng vào của hệ thống, nằm trên cùng, phía trên Application Layer.
- Không chứa business logic, chỉ nhận request, trả response, chuyển dữ liệu giữa client và các layer bên dưới.

Controllers/

- Chứa: Các class nhận HTTP/gRPC request từ client, trả response.
- Ví dụ:
  - AppointmentsController: Nhận request đặt lịch, hủy lịch, lấy danh sách lịch hẹn.
  - PaymentController: Nhận request thanh toán, hoàn tiền.
  - BillingsController: Nhận request tạo hóa đơn, lấy chi tiết hóa đơn.
- Ý nghĩa:
  - Chỉ nhận request, validate sơ bộ, chuyển dữ liệu sang Application Layer.

- Không chứa logic nghiệp vụ, không truy xuất DB trực tiếp.

Services/

- Chứa: Các service hỗ trợ cho controller, thường là gRPC service hoặc các adapter cho external API.
- Ví dụ: AppointmentGrpcService: Adapter cho gRPC endpoint lịch hẹn.
- Ý nghĩa: Giúp hệ thống mở rộng thêm các giao thức (gRPC, REST, ...).

Protos/

- Chứa: Định nghĩa các contract gRPC (protobuf).
- Ý nghĩa: Giúp các service giao tiếp qua gRPC, dễ tích hợp với hệ thống khác.

Luồng đi của API Layer

- Client gửi request (HTTP/gRPC) đến controller.
- Controller nhận request, validate sơ bộ, chuyển dữ liệu sang Application Layer (thường qua service hoặc use-case handler).
- Application Layer xử lý nghiệp vụ, gọi các interface của Domain, orchestrate các bước.
- Kết quả trả về cho Controller, controller trả response cho client.

Ví dụ cụ thể: (BillingsController)

Nhận request từ client:

Ví dụ: POST /api/billings/{id}/process-payment để thanh toán hóa đơn.

```
[HttpPost("{id}/process-payment")]
public async Task<ActionResult<PaymentResultDto>> ProcessPayment(int
id, ProcessPaymentRequest request)
{
  try
  {
     var currentUserId = GetCurrentUserId();
     var currentUserRole = GetCurrentUserRole();

     var billing = await _billingRepository.GetByIdAsync(id);
```

```
    if (billing == null)
    {
        return NotFound($"Billing with ID {id} not found");
    }

    if (!CanAccessBilling(billing, currentUserId, currentUserRole))
    {
        return Forbid("You don't have permission to process this payment");
    }

    if (billing.Status != "Pending")
    {
        return BadRequest($"Cannot process payment for billing with status:
{billing.Status}");
    }

    _logger.LogInformation("Processing payment for billing {BillingId} using
{PaymentMethod}",
        id, billing.PaymentMethod);

    // Use factory to create payment method
    var paymentMethod = _paymentFactory.CreatePaymentMethod(billing.
PaymentMethod);

    var paymentRequest = new PaymentRequest
    {
        Amount = billing.TotalAmount, // Use strategy-calculated amount
        Currency = request.Currency ?? "USD",
        Description = billing.Description ?? $"Payment for appointment {billin
g.AppointmentId}",
        PatientId = billing.PatientId,
        AppointmentId = billing.AppointmentId,
        AdditionalData = request.AdditionalData ?? new Dictionary<string, ob
ject>()
    };

    // Add billing context to additional data
    paymentRequest.AdditionalData["billing_id"] = billing.Id;
```

```
    paymentRequest.AdditionalData["billing_type"] = billing.BillingType;
    paymentRequest.AdditionalData["invoice_number"] = billing.InvoiceNu
mber ?? "";

    var paymentResult = await paymentMethod.ProcessPaymentAsync(pay
mentRequest);

    // For Stripe, don't mark as completed immediately (will be done via we
bhook)
    // For other methods like Cash, mark as completed
    if (billing.PaymentMethod.Equals("Stripe", StringComparison.OrdinalIgn
oreCase))
    {
        billing.Status = paymentResult.IsSuccess ? "Processing" : "Failed";
    }
    else
    {
        billing.Status = paymentResult.IsSuccess ? "Completed" : "Failed";
    }

    billing.TransactionId = paymentResult.TransactionId;
    billing.FailureReason = paymentResult.FailureReason;
    billing.UpdatedAt = DateTime.UtcNow;

    await _billingRepository.UpdateAsync(billing);

    // Publish events to RabbitMQ
    if (paymentResult.IsSuccess)
    {
        if (billing.PaymentMethod.Equals("Stripe", StringComparison.OrdinalI
gnoreCase))
        {
            await _rabbitMQService.PublishPaymentInitiatedAsync(new Payme
ntInitiatedEvent
            {
                BillingId = billing.Id,
                AppointmentId = billing.AppointmentId,
                PatientId = billing.PatientId,
```

```
                Amount = billing.TotalAmount,
                PaymentMethod = billing.PaymentMethod,
                SessionId = billing.TransactionId,
                InitiatedAt = paymentResult.ProcessedAt,
                CheckoutUrl = paymentResult.AdditionalData.GetValueOrDefault
("checkout_url") as string
            });
        }
        else
        {
            await _rabbitMQService.PublishPaymentProcessedAsync(new Pay
mentProcessedEvent
            {
                BillingId = billing.Id,
                AppointmentId = billing.AppointmentId,
                PatientId = billing.PatientId,
                Amount = billing.TotalAmount,
                PaymentMethod = billing.PaymentMethod,
                TransactionId = billing.TransactionId,
                ProcessedAt = paymentResult.ProcessedAt,
                ProcessedByUserId = currentUserId,
                ProcessedByRole = currentUserRole
            });
        }
    }
    else
    {
        await _rabbitMQService.PublishPaymentFailedAsync(new PaymentFa
iledEvent
        {
            BillingId = billing.Id,
            AppointmentId = billing.AppointmentId,
            PatientId = billing.PatientId,
            Amount = billing.TotalAmount,
            PaymentMethod = billing.PaymentMethod,
            FailureReason = billing.FailureReason,
            FailedAt = paymentResult.ProcessedAt,
            ProcessedByUserId = currentUserId,
```

```
            ProcessedByRole = currentUserRole
        });
    }

    var resultDto = new PaymentResultDto
    {
        IsSuccess = paymentResult.IsSuccess,
        TransactionId = paymentResult.TransactionId,
        FailureReason = paymentResult.FailureReason,
        Amount = paymentResult.Amount,
        ProcessedAt = paymentResult.ProcessedAt,
        AdditionalData = paymentResult.AdditionalData // This will include ch
eckout_url for Stripe
    };

    _logger.LogInformation("Payment processing completed for billing {Billi
ngId}: {Success}",
        id, paymentResult.IsSuccess ? "Success" : "Failed");

    return Ok(resultDto);
 }
 catch (NotSupportedException ex)
 {
    _logger.LogError(ex, "Unsupported payment method for billing {BillingI
d}", id);
    return BadRequest(ex.Message);
 }
 catch (Exception ex)
 {
    _logger.LogError(ex, "Error processing payment for billing {BillingId}", i
d);
    return StatusCode(500, "Internal server error");
 }
}
```

Luồng xử lý thanh toán (ProcessPayment):

- Nhận request từ client:

Client gọi API /api/billings/{id}/process-payment với thông tin billing và payment.

- Lấy thông tin billing:

Controller gọi repository để lấy billing theo id.

- Kiểm tra quyền truy cập:

Kiểm tra user hiện tại có quyền xử lý billing này không (Admin, Doctor, hoặc đúng Patient).

- Kiểm tra trạng thái billing:

Chỉ cho phép thanh toán khi billing đang ở trạng thái "Pending".

- Tạo payment method:

Dùng factory để lấy đúng phương thức thanh toán (Stripe, Cash, ...).

- Chuẩn bị PaymentRequest:

Tạo object chứa thông tin cần thiết cho thanh toán.

- Thực hiện thanh toán:

Gọi phương thức thanh toán thực tế (ở tầng Infrastructure thông qua interface ở tầng domain).

- Cập nhật trạng thái billing:

Nếu Stripe thì chuyển sang "Processing", nếu Cash thì "Completed", nếu lỗi thì "Failed".

- Lưu billing vào database:

Cập nhật lại billing với trạng thái mới (ở tầng Infrastructure thông qua interface ở tầng domain).

- Publish event lên RabbitMQ:

Gửi event PaymentInitiated/PaymentProcessed/PaymentFailed để các service khác xử lý tiếp (notification, audit, ...). (ở tầng Infrastructure thông qua interface ở tầng domain).

- Trả kết quả về cho client:

Trả về thông tin kết quả thanh toán (thành công/thất bại, transactionId, lý do, ...).

Summary

Luồng dữ liệu tổng quát

Client → API Layer (Controller nhận request)

→ Application Layer (Use-case orchestration, validation, transaction)

→ Domain Layer (Entities, business rules, domain events)

→ Infrastructure Layer (DB, MQ, cache, external API implement)

→ Quay ngược lại theo chiều trả kết quả → Client.

Kiến trúc 4 tầng (API, Application, Domain, Infrastructure) giúp hệ thống:

- Tách biệt rõ ràng giữa logic nghiệp vụ và hạ tầng kỹ thuật.
- Dễ mở rộng: chỉ cần bổ sung implement mới ở Infrastructure hoặc use-case mới ở Application mà không ảnh hưởng đến core domain.
- Dễ bảo trì & test: nghiệp vụ được cô lập, có thể mock/replace các phần kỹ thuật khi kiểm thử.
- Linh hoạt công nghệ: dễ dàng thay thế database, message broker, cache, storage mà không thay đổi business logic.
- Hỗ trợ kiến trúc hướng sự kiện (event-driven): mọi hành động quan trọng (appointment created, payment processed...) được ghi nhận qua domain event và publish ra MQ để các service khác xử lý.