ASSIGNMENT 01

Hà Mạnh Dũng
MSV: B22DCCN125

Ngày 19 tháng 1 năm 2026

1 Tổng Quan

1.1 Giới Thiệu Dự Án

Dự án Book Store Web System được phát triển nhằm minh họa ba kiến trúc phần mềm khác nhau:

- **Monolithic Architecture**: Kiến trúc đơn thể, tất cả chức năng trong một ứng dụng
- **Clean Architecture**: Kiến trúc sạch với các lớp tách biệt rõ ràng
- **Microservices Architecture**: Kiến trúc vi dịch vụ với các service độc lập

1.2 Điểm Chung Giữa Các Version

1.2.1 Entities và Database Schema

Cả ba version đều sử dụng cùng một mô hình dữ liệu với 4 entities chính:

1. **Customer**: Quản lý thông tin khách hàng
   - `id`: Khóa chính
   - `name`: Tên khách hàng
   - `email`: Email (unique)
   - `password`: Mật khẩu đã được hash
2. **Book**: Quản lý thông tin sách
   - `id`: Khóa chính
   - `title`: Tiêu đề sách
   - `author`: Tác giả
   - `price`: Giá tiền
   - `stock`: Số lượng tồn kho
3. **Cart**: Quản lý giỏ hàng

- `id`: Khóa chính
- `customer_id`: Tham chiếu đến Customer
- `created_at`: Thời gian tạo

4. **CartItem**: Quản lý các sản phẩm trong giỏ hàng
    - `id`: Khóa chính
    - `cart_id`: Tham chiếu đến Cart
    - `book_id`: Tham chiếu đến Book
    - `quantity`: Số lượng

1.2.2 Functional Requirements

Cả ba version đều implement các chức năng sau:

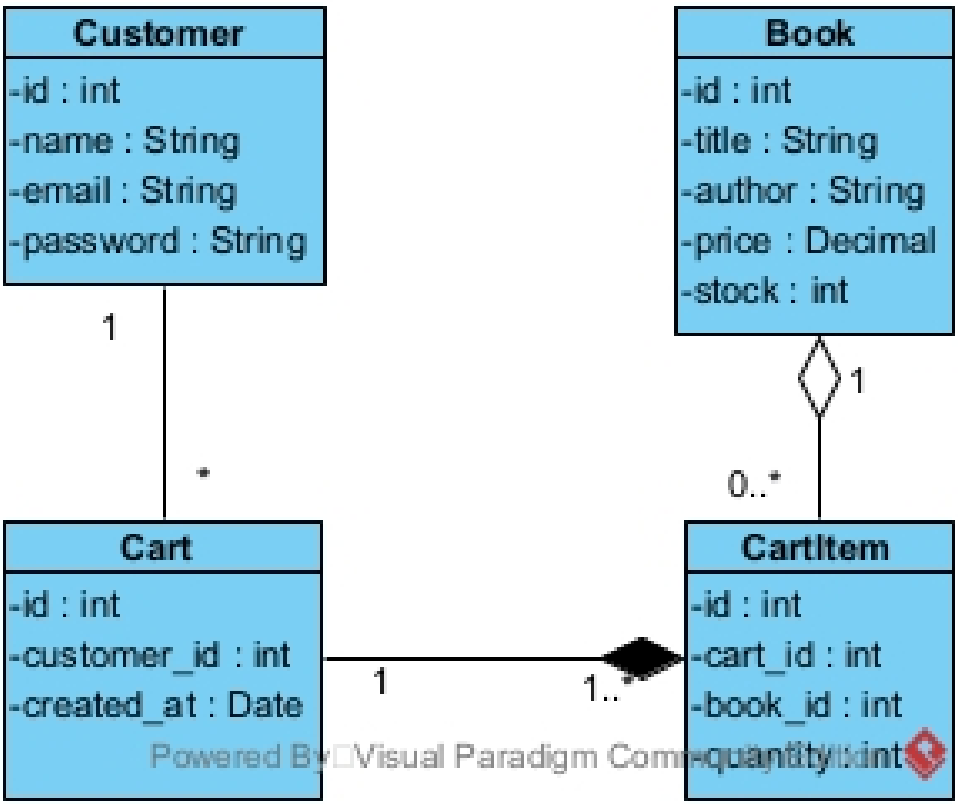
- **Customer Registration**: Đăng ký tài khoản khách hàng mới
- **Customer Login**: Đăng nhập với email và password
- **View Book Catalog**: Xem danh sách tất cả sách
- **Add to Cart**: Thêm sách vào giỏ hàng
- **View Cart**: Xem nội dung giỏ hàng

1.2.3 Technology Stack

- **Backend Framework**: Django 4.2.7
- **Database**: MySQL
- **API Framework**: Django REST Framework
- **Authentication**: Django password hashing
- **CORS**: django-cors-headers

1.2.4 Class Diagram

Cả ba version đều sử dụng cùng một Class Diagram để mô tả mối quan hệ giữa các entities:



Hình 1: Class Diagram - Mối quan hệ giữa các entities

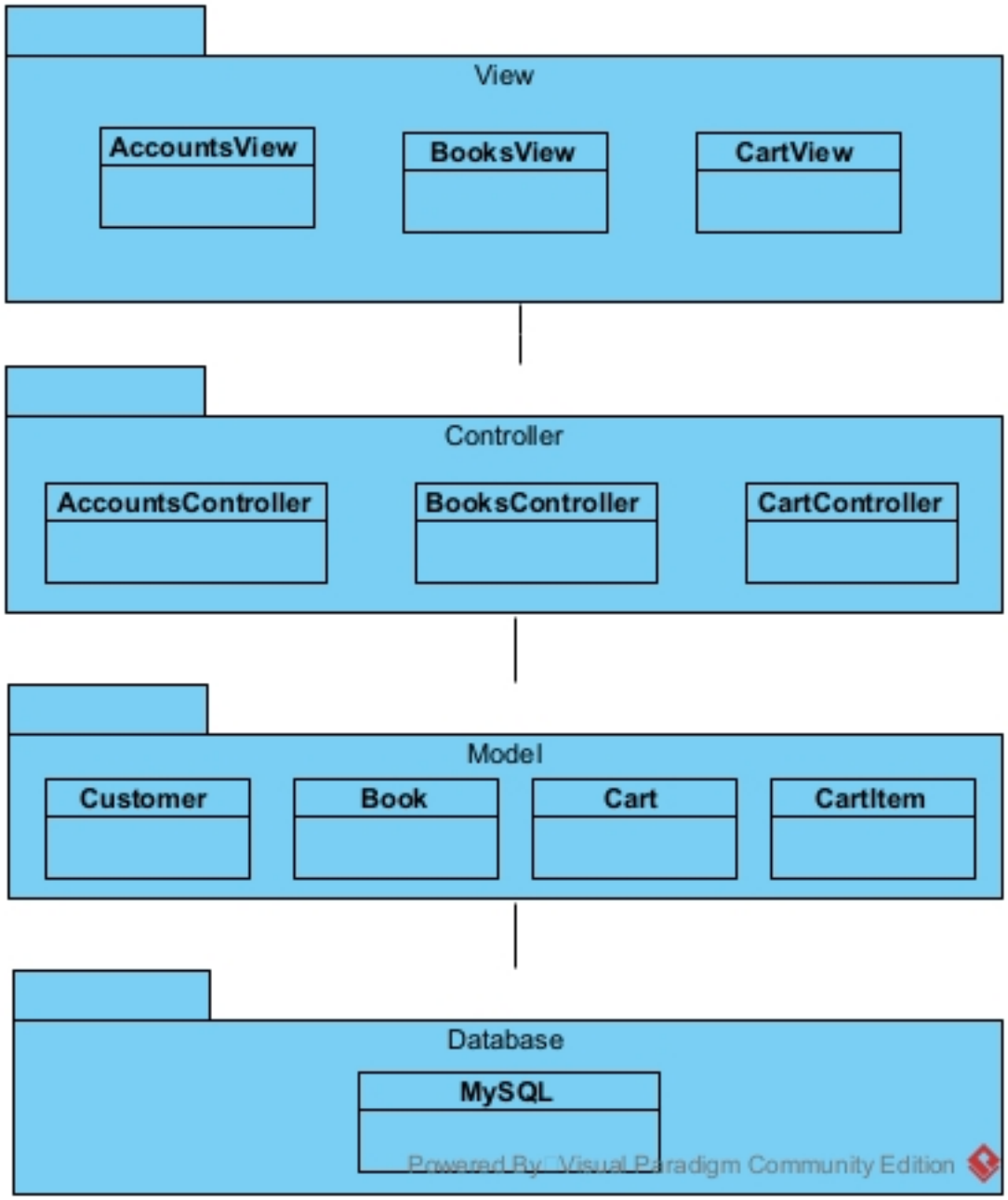
Giải thích các mối quan hệ:

- **Customer → Cart**: Association (1 to 0..*) - Một khách hàng có thể có nhiều giỏ hàng
- **Cart → CartItem**: Composition (1 to 1..*) - CartItem không thể tồn tại độc lập, phụ thuộc vào Cart
- **Book → CartItem**: Aggregation (1 to 0..*) - Book có thể tồn tại độc lập, CartItem chỉ tham chiếu đến Book

2 Version A - Monolithic Architecture

2.1 Tổng Quan Kiến Trúc

Kiến trúc Monolithic là kiến trúc truyền thống, tất cả các chức năng được đặt trong một ứng dụng Django duy nhất. Hệ thống được tổ chức theo mô hình MVC (Model-View-Controller).



Hình 2: MVC Layer Diagram - Monolithic Architecture

2.2 Cấu Trúc Dự Án

Dự án được tổ chức thành các Django apps:

- `accounts/`: Quản lý khách hàng (registration, login)

- `books/`: Quản lý danh mục sách
- `cart/`: Quản lý giỏ hàng
- `bookstore/`: Cấu hình chính của Django project

2.3 Giải Thích Code

2.3.1 Models Layer

Customer Model (`accounts/models.py`):

```
//Ha Manh Dung - B22DCCN125
class Customer(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=255)
    email = models.EmailField(unique=True)
    password = models.CharField(max_length=255)

    def set_password(self, raw_password):
        self.password = make_password(raw_password)

    def check_password(self, raw_password):
        return check_password(raw_password, self.password)
```

Giải thích:

- Model Customer định nghĩa cấu trúc bảng `customers` trong database
- `set_password()`: Hash password trước khi lưu vào database
- `check_password()`: Xác thực password khi đăng nhập

Cart và CartItem Models (`cart/models.py`):

```
//Ha Manh Dung - B22DCCN125
class Cart(models.Model):
    id = models.AutoField(primary_key=True)
    customer = models.ForeignKey(Customer,
                                 on_delete=models.CASCADE,
                                 related_name='carts')
    created_at = models.DateTimeField(auto_now_add=True)

class CartItem(models.Model):
    id = models.AutoField(primary_key=True)
    cart = models.ForeignKey(Cart,
                             on_delete=models.CASCADE,
                             related_name='items')
    book = models.ForeignKey(Book,
                             on_delete=models.CASCADE)
    quantity = models.IntegerField(default=1)
```

Giải thích:

- `Cart` có ForeignKey đến `Customer` với CASCADE delete

- `CartItem` có ForeignKey đến cả `Cart` và `Book`
- Khi Cart bị xóa, tất cả CartItem liên quan cũng bị xóa (CASCADE)

2.3.2 Views Layer (Controller)

Customer Registration (`accounts/views.py`):

```
//Ha Manh Dung - B22DCCN125
@api_view(['POST'])
def register(request):
    serializer = CustomerSerializer(data=request.data)
    if serializer.is_valid():
        customer = serializer.save()
        response_serializer = CustomerResponseSerializer(customer)
        return Response(response_serializer.data,
                        status=status.HTTP_201_CREATED)
    return Response(serializer.errors,
                    status=status.HTTP_400_BAD_REQUEST)
```

Giải thích:

- Sử dụng decorator `@api_view` để xử lý HTTP POST request
- `CustomerSerializer` validate và tạo Customer mới
- `CustomerResponseSerializer` trả về dữ liệu không bao gồm password
- Trả về HTTP 201 nếu thành công, 400 nếu có lỗi validation

Add to Cart (`cart/views.py`):

```
//Ha Manh Dung - B22DCCN125
@api_view(['POST'])
def add_to_cart(request, customer_id):
    # Validate customer exists
    try:
        customer = Customer.objects.get(id=customer_id)
    except Customer.DoesNotExist:
        return Response({'error': 'Customer not found'},
                        status=status.HTTP_404_NOT_FOUND)

    # Validate book exists and stock
    book = Book.objects.get(id=book_id)
    if book.stock < quantity:
        return Response({'error': 'Insufficient stock'},
                        status=status.HTTP_400_BAD_REQUEST)

    # Get or create cart
    cart, created = Cart.objects.get_or_create(customer=customer)

    # Get or create cart item
    cart_item, item_created = CartItem.objects.get_or_create(
        cart=cart, book=book, defaults={'quantity': quantity}
```

```
    )

    if not item_created:
        cart_item.quantity += quantity
        cart_item.save()
```

Giải thích:

- Validate customer và book tồn tại trước khi thêm vào cart
- Kiểm tra số lượng tồn kho đủ không
- Sử dụng `get_or_create()` để tạo cart nếu chưa có
- Nếu CartItem đã tồn tại, tăng quantity thay vì tạo mới

2.3.3 Serializers Layer

CustomerSerializer (`accounts/serializers.py`):

```
//Ha Manh Dung - B22DCCN125
class CustomerSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True,
       required=True)

    class Meta:
        model = Customer
        fields = ['id', 'name', 'email', 'password']
        extra_kwargs = {
            'password': {'write_only': True}
        }

    def create(self, validated_data):
        customer = Customer.objects.create(
            name=validated_data['name'],
            email=validated_data['email']
        )
        customer.set_password(validated_data['password'])
        customer.save()
        return customer
```

Giải thích:

- `write_only=True`: Password chỉ nhận vào, không trả về trong response
- Override method `create()` để hash password trước khi lưu
- Sử dụng `set_password()` để đảm bảo password được hash đúng cách

2.3.4 URL Routing

Main URLs (`bookstore/urls.py`):

```
//Ha Manh Dung - B22DCCN125
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/accounts/', include('accounts.urls')),
    path('api/books/', include('books.urls')),
    path('api/cart/', include('cart.urls')),
]
```

Giải thích:

- Tổ chức URLs theo từng app
- Mỗi app có file `urls.py` riêng để quản lý routing
- Sử dụng `include()` để include các URL patterns từ apps

2.4 Ưu và Nhược Điểm

Ưu điểm:

- Đơn giản, dễ phát triển ban đầu
- Dễ test và debug
- Không cần xử lý inter-service communication
- Phù hợp với ứng dụng nhỏ và vừa

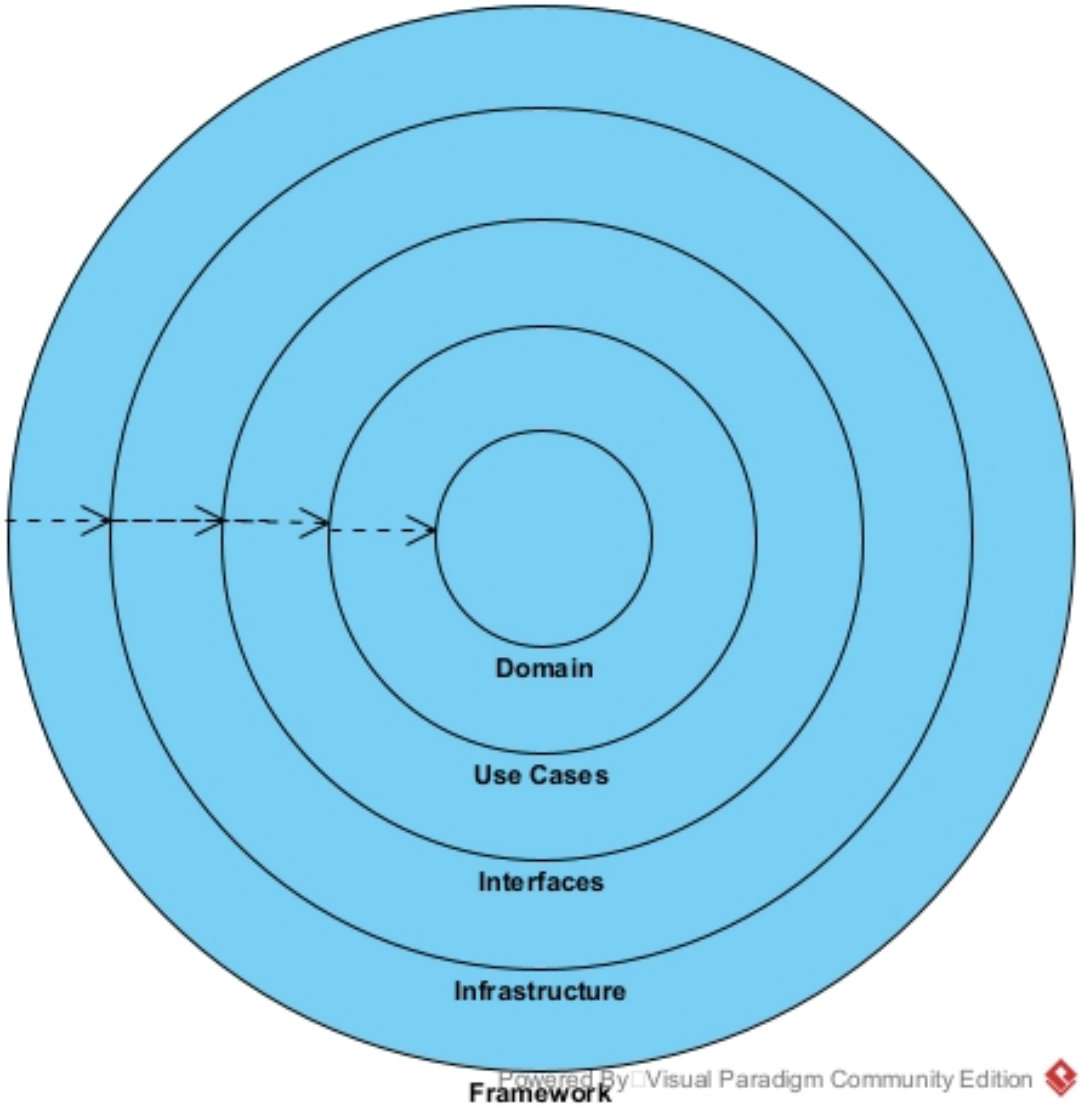
Nhược điểm:

- Khó scale khi ứng dụng lớn
- Tất cả code trong một codebase, khó maintain
- Một lỗi có thể ảnh hưởng toàn bộ hệ thống
- Khó deploy từng phần riêng biệt

3 Version B - Clean Architecture

3.1 Tổng Quan Kiến Trúc

Clean Architecture tách biệt business logic khỏi framework và infrastructure. Kiến trúc này tuân theo Dependency Inversion Principle, các dependencies chỉ hướng vào trong.



Hình 3: Clean Architecture Diagram

3.2 Cấu Trúc Dự Án

Dự án được tổ chức thành 5 layers:

- `domain/`: Entities và Repository interfaces (trung tâm)
- `usecases/`: Business logic use cases
- `interfaces/`: Serializers và adapters
- `infrastructure/`: Repository implementations

- `framework/`: Django framework layer (ngoài cùng)

3.3 Giải Thích Code

3.3.1 Domain Layer

Entities (`domain/entities.py`):

```python
//Ha Manh Dung - B22DCCN125
@dataclass
class Customer:
    """Customer entity"""
    id: Optional[int]
    name: str
    email: str
    password: str

    def __post_init__(self):
        if self.id is None:
            self.id = 0
```

Giải thích:

- Sử dụng `@dataclass` để định nghĩa entities đơn giản
- Entities không phụ thuộc vào framework hay database
- Chứa pure business objects, không có logic phức tạp

Repository Interfaces (`domain/repositories.py`):

```python
//Ha Manh Dung - B22DCCN125
class ICustomerRepository(ABC):
    """Customer repository interface"""

    @abstractmethod
    def create(self, customer: Customer) -> Customer:
        pass

    @abstractmethod
    def get_by_id(self, customer_id: int) -> Optional[Customer]:
        pass

    @abstractmethod
    def get_by_email(self, email: str) -> Optional[Customer]:
        pass
```

Giải thích:

- Định nghĩa contracts (interfaces) cho data access
- Sử dụng `ABC` (Abstract Base Class) để đảm bảo implementation
- Domain layer không biết cách implement, chỉ biết interface

3.3.2 Use Cases Layer

RegisterCustomerUseCase (usecases/customer_usecases.py):

```python
//Ha Manh Dung - B22DCCN125
class RegisterCustomerUseCase:
    """Register a new customer"""

    def __init__(self, customer_repository: ICustomerRepository):
        self.customer_repository = customer_repository

    def execute(self, name: str, email: str, password: str) ->
       Customer:
        # Check if email already exists
        existing = self.customer_repository.get_by_email(email)
        if existing:
            raise ValueError("Email already exists")

        # Create customer with hashed password
        hashed_password = make_password(password)
        customer = Customer(
            id=None, name=name, email=email,
               password=hashed_password
        )

        return self.customer_repository.create(customer)
```

Giải thích:

- Use case nhận repository interface, không phụ thuộc vào implementation
- Chứa business logic: kiểm tra email trùng, hash password
- Có thể test độc lập mà không cần Django
- Method `execute()` thực hiện một use case cụ thể

AddToCartUseCase (usecases/cart_usecases.py):

```python
//Ha Manh Dung - B22DCCN125
class AddToCartUseCase:
    def __init__(self, cart_repo, cart_item_repo,
                 customer_repo, book_repo):
        self.cart_repository = cart_repo
        self.cart_item_repository = cart_item_repo
        self.customer_repository = customer_repo
        self.book_repository = book_repo

    def execute(self, customer_id: int, book_id: int,
                quantity: int) -> Cart:
        # Validate customer exists
        customer = self.customer_repository.get_by_id(customer_id)
        if not customer:
            raise ValueError("Customer not found")
```

```
16
17          # Validate book exists and has stock
18          book = self.book_repository.get_by_id(book_id)
19          if not book:
20              raise ValueError("Book not found")
21          if book.stock < quantity:
22              raise ValueError("Insufficient stock")
23
24          # Get or create cart
25          cart = self.cart_repository.get_or_create_by_customer_id(
26              customer_id
27          )
28
29          # Get or create cart item
30          cart_item = self.cart_item_repository.get_or_create(
31              cart_id=cart.id, book_id=book_id, quantity=quantity
32          )
33
34          return cart
```

Giải thích:

- Use case phức tạp hơn, cần nhiều repositories
- Tập trung business logic: validation, business rules
- Throw exceptions thay vì trả về error codes
- Dễ test vì không phụ thuộc vào framework

3.3.3 Infrastructure Layer

CustomerRepository (infrastructure/repositories.py):

```
1  //Ha Manh Dung - B22DCCN125
2  class CustomerRepository(ICustomerRepository):
3      """Customer repository implementation"""
4
5      def create(self, customer: Customer) -> Customer:
6          model = CustomerModel.objects.create(
7              name=customer.name,
8              email=customer.email,
9              password=customer.password
10         )
11         return Customer(
12             id=model.id,
13             name=model.name,
14             email=model.email,
15             password=model.password
16         )
17
18     def get_by_email(self, email: str) -> Optional[Customer]:
19         try:
```

```
            model = CustomerModel.objects.get(email=email)
            return Customer(
                id=model.id,
                name=model.name,
                email=model.email,
                password=model.password
            )
        except CustomerModel.DoesNotExist:
            return None
```

Giải thích:

- Implement interface từ domain layer
- Sử dụng Django ORM (`CustomerModel`) để truy cập database
- Chuyển đổi giữa Domain entities và Django models
- Xử lý exceptions và trả về None nếu không tìm thấy

3.3.4 Framework Layer

Views (`framework/views.py`):

```
//Ha Manh Dung - B22DCCN125
# Initialize repositories
customer_repo = CustomerRepository()
book_repo = BookRepository()
cart_repo = CartRepository()
cart_item_repo = CartItemRepository()

@api_view(['POST'])
def register(request):
    try:
        name = request.data.get('name')
        email = request.data.get('email')
        password = request.data.get('password')

        if not all([name, email, password]):
            return Response(
                {'error': 'Missing required fields'},
                status=status.HTTP_400_BAD_REQUEST
            )

        use_case = RegisterCustomerUseCase(customer_repo)
        customer = use_case.execute(name, email, password)

        return Response(
            CustomerSerializer.to_dict(customer),
            status=status.HTTP_201_CREATED
        )
    except ValueError as e:
        return Response(
```

```
            {'error': str(e)},
            status=status.HTTP_400_BAD_REQUEST
        )
```

Giải thích:

- Views là entry point, xử lý HTTP requests
- Khởi tạo repositories và use cases
- Gọi use case để thực hiện business logic
- Chuyển đổi entities thành JSON response qua serializers
- Xử lý exceptions và trả về HTTP status codes phù hợp

3.4 Dependency Flow

Dependencies chỉ hướng vào trong:

1. **Framework** → Infrastructure, Interfaces, Use Cases, Domain
2. **Infrastructure** → Domain (implements interfaces)
3. **Interfaces** → Use Cases, Domain
4. **Use Cases** → Domain
5. **Domain** → Không phụ thuộc gì

3.5 Ưu và Nhược Điểm

Ưu điểm:

- Business logic độc lập với framework
- Dễ test, có thể test use cases không cần Django
- Dễ maintain, thay đổi framework không ảnh hưởng business logic
- Tuân theo SOLID principles

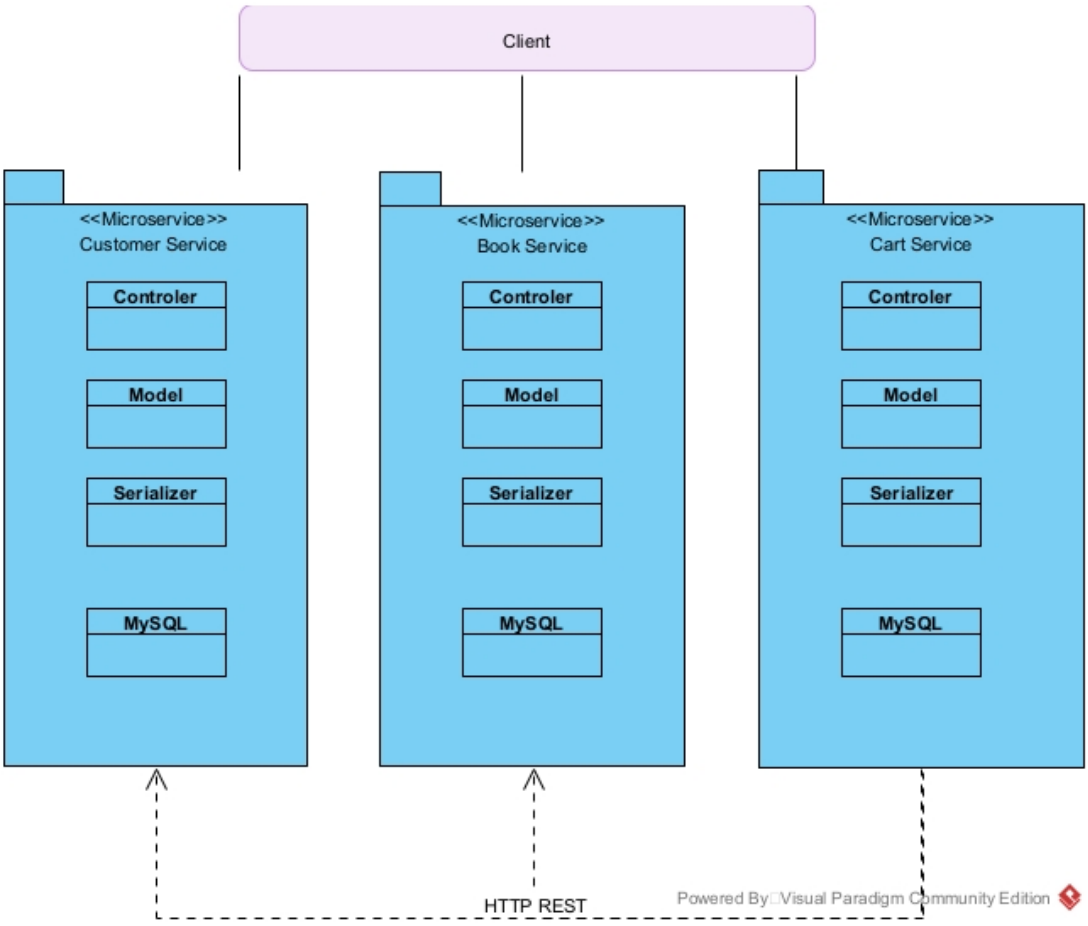
Nhược điểm:

- Phức tạp hơn Monolithic
- Nhiều layers, nhiều files
- Có thể over-engineering cho dự án nhỏ
- Cần hiểu rõ Clean Architecture principles

4 Version C - Microservices Architecture

4.1 Tổng Quan Kiến Trúc

Microservices Architecture chia hệ thống thành các services độc lập, mỗi service có database riêng và giao tiếp qua REST APIs.



Hình 4: Microservices Architecture Diagram

4.2 Các Services

Hệ thống được chia thành 3 services:

1. **Customer Service** (Port 8001): Quản lý khách hàng
2. **Book Service** (Port 8002): Quản lý danh mục sách
3. **Cart Service** (Port 8003): Quản lý giỏ hàng

4.3 Giải Thích Code

4.3.1 Customer Service

Model (`customers/models.py`):

```python
//Ha Manh Dung - B22DCCN125
class Customer(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=255)
    email = models.EmailField(unique=True)
    password = models.CharField(max_length=255)

    def set_password(self, raw_password):
        self.password = make_password(raw_password)

    def check_password(self, raw_password):
        return check_password(raw_password, self.password)
```

View (customers/views.py):

```python
//Ha Manh Dung - B22DCCN125
@api_view(['GET'])
def get_customer(request, customer_id):
    """Get customer by ID (for other services)"""
    try:
        customer = Customer.objects.get(id=customer_id)
        serializer = CustomerResponseSerializer(customer)
        return Response(serializer.data,
           status=status.HTTP_200_OK)
    except Customer.DoesNotExist:
        return Response({'error': 'Customer not found'},
                        status=status.HTTP_404_NOT_FOUND)
```

Giải thích:

- Customer Service có endpoint `GET /api/customers/<id>/` để các service khác gọi
- Trả về thông tin customer dạng JSON
- Sử dụng cho inter-service communication

4.3.2 Book Service

View (books/views.py):

```python
//Ha Manh Dung - B22DCCN125
@api_view(['GET'])
def get_books_by_ids(request):
    """Get multiple books by IDs (for other services)"""
    book_ids = request.GET.getlist('ids')
    if not book_ids:
        return Response({'error': 'No book IDs provided'},
                        status=status.HTTP_400_BAD_REQUEST)

    try:
        book_ids = [int(id) for id in book_ids]
        books = Book.objects.filter(id__in=book_ids)
        serializer = BookSerializer(books, many=True)
```

```
        return Response(serializer.data,
            status=status.HTTP_200_OK)
    except ValueError:
        return Response({'error': 'Invalid book IDs'},
                        status=status.HTTP_400_BAD_REQUEST)
```

Giải thích:

- Endpoint `GET /api/books/batch/?ids=1&ids=2` để lấy nhiều books cùng lúc
- Tối ưu cho trường hợp Cart Service cần lấy thông tin nhiều books
- Giảm số lượng HTTP requests

4.3.3 Cart Service

Service Clients (carts/services.py):

```
//Ha Manh Dung - B22DCCN125
class CustomerService:
    """Client for customer-service"""

    @staticmethod
    def get_customer(customer_id):
        """Get customer from customer-service"""
        try:
            url =
                f"{settings.CUSTOMER_SERVICE_URL}/api/customers/{customer_id}/
            response = requests.get(url, timeout=5)
            if response.status_code == 200:
                return response.json()
            return None
        except requests.RequestException:
            return None

class BookService:
    """Client for book-service"""

    @staticmethod
    def get_book(book_id):
        """Get book from book-service"""
        try:
            url =
                f"{settings.BOOK_SERVICE_URL}/api/books/{book_id}/"
            response = requests.get(url, timeout=5)
            if response.status_code == 200:
                return response.json()
            return None
        except requests.RequestException:
            return None

    @staticmethod
    def get_books_by_ids(book_ids):
```

```
        """Get multiple books from book-service"""
        try:
            url = f"{settings.BOOK_SERVICE_URL}/api/books/batch/"
            params = [('ids', book_id) for book_id in book_ids]
            response = requests.get(url, params=params, timeout=5)
            if response.status_code == 200:
                return response.json()
            return []
        except requests.RequestException:
            return []
```

Giải thích:

- Service clients là HTTP clients để giao tiếp với các services khác
- Sử dụng thư viện `requests` để gọi REST APIs
- Có timeout để tránh blocking quá lâu
- Xử lý exceptions và trả về None/[] nếu có lỗi

Add to Cart View (`carts/views.py`):

```
//Ha Manh Dung - B22DCCN125
@api_view(['POST'])
def add_to_cart(request, customer_id):
    """Add books to the shopping cart"""
    # Validate customer exists via customer-service
    customer_data = CustomerService.get_customer(customer_id)
    if not customer_data:
        return Response({'error': 'Customer not found'},
                        status=status.HTTP_404_NOT_FOUND)

    # Validate book exists and has stock via book-service
    book_data = BookService.get_book(book_id)
    if not book_data:
        return Response({'error': 'Book not found'},
                        status=status.HTTP_404_NOT_FOUND)

    if book_data['stock'] < quantity:
        return Response({'error': 'Insufficient stock'},
                        status=status.HTTP_400_BAD_REQUEST)

    # Get or create cart for customer
    cart, created = Cart.objects.get_or_create(
        customer_id=customer_id
    )

    # Get or create cart item
    cart_item, item_created = CartItem.objects.get_or_create(
        cart=cart, book_id=book_id, defaults={'quantity':
           quantity}
    )
```

```
30
31      # Fetch cart items with book data
32      items = CartItem.objects.filter(cart=cart)
33      book_ids = [item.book_id for item in items]
34      books_data = BookService.get_books_by_ids(book_ids)
35      books_dict = {book['id']: book for book in books_data}
36
37      # Attach book data to items
38      for item in items:
39          item._book_data = books_dict.get(item.book_id)
40
41      cart._customer_data = customer_data
42
43      serializer = CartSerializer(cart)
44      return Response(serializer.data, status=status.HTTP_200_OK)
```

Giải thích:

- Cart Service gọi Customer Service để validate customer
- Gọi Book Service để validate book và kiểm tra stock
- Lưu cart và cart items vào database riêng của Cart Service
- Fetch thông tin books từ Book Service để trả về đầy đủ
- Attach data vào objects để serializer có thể serialize

4.3.4 Database per Service

Mỗi service có database riêng:

- `bookstore_customer`: Database cho Customer Service
- `bookstore_book`: Database cho Book Service
- `bookstore_cart`: Database cho Cart Service

Giải thích:

- Mỗi service độc lập về dữ liệu
- Có thể scale từng service riêng biệt
- Không có shared database, tránh tight coupling
- Mỗi service có thể chọn database phù hợp

4.4 Inter-Service Communication

- **Cart Service → Customer Service**:
  - Purpose: Validate customer exists
  - Method: HTTP GET
  - Endpoint: `http://localhost:8001/api/customers/<id>/`
- **Cart Service → Book Service**:
  - Purpose: Get book information and validate stock
  - Method: HTTP GET
  - Endpoints:
    * `http://localhost:8002/api/books/<id>/`
    * `http://localhost:8002/api/books/batch/?ids=1&ids=2`

4.5 Ưu và Nhược Điểm

Ưu điểm:

- Mỗi service có thể phát triển và deploy độc lập
- Có thể scale từng service riêng biệt
- Fault isolation: lỗi một service không ảnh hưởng service khác
- Có thể sử dụng công nghệ khác nhau cho từng service
- Phù hợp với team lớn, mỗi team phụ trách một service

Nhược điểm:

- Phức tạp hơn về infrastructure và deployment
- Cần xử lý distributed transactions
- Network latency giữa các services
- Khó debug khi có lỗi liên quan đến nhiều services
- Cần service discovery và load balancing

5 So Sánh Ba Kiến Trúc

5.1 Bảng So Sánh

Tiêu chí	Monolithic	Clean Architecture	Microservices
Độ phức tạp	Thấp	Trung bình	Cao
Dễ phát triển	Rất dễ	Dễ	Khó
Dễ test	Dễ	Rất dễ	Khó
Khả năng scale	Thấp	Trung bình	Cao
Maintainability	Trung bình	Cao	Trung bình
Deployment	Đơn giản	Đơn giản	Phức tạp
Fault isolation	Thấp	Trung bình	Cao
Technology diversity	Không	Không	Có

Bảng 1: So sánh ba kiến trúc

5.2 Khi Nào Sử Dụng

Monolithic Architecture:

- Ứng dụng nhỏ và vừa
- Team nhỏ
- Cần phát triển nhanh
- Không cần scale cao

Clean Architecture:

- Ứng dụng cần maintain lâu dài
- Cần business logic phức tạp
- Có thể thay đổi framework trong tương lai
- Cần test coverage cao

Microservices Architecture:

- Ứng dụng lớn, nhiều teams
- Cần scale cao
- Các services có thể scale độc lập
- Cần fault isolation
- Có thể sử dụng công nghệ khác nhau

6 Kết Luận

Dự án Book Store Web System đã thành công implement ba kiến trúc phần mềm khác nhau:

1. **Monolithic Architecture**: Đơn giản, phù hợp cho ứng dụng nhỏ
2. **Clean Architecture**: Tách biệt business logic, dễ maintain và test
3. **Microservices Architecture**: Phù hợp cho hệ thống lớn, có thể scale độc lập

Mỗi kiến trúc có ưu nhược điểm riêng và phù hợp với các tình huống khác nhau. Việc hiểu rõ từng kiến trúc giúp lựa chọn giải pháp phù hợp cho từng dự án cụ thể.

7 Tài Liệu Tham Khảo

- Django Documentation: `https://docs.djangoproject.com/`
- Django REST Framework: `https://www.django-rest-framework.org/`
- Clean Architecture - Robert C. Martin
- Microservices Patterns - Chris Richardson
- MySQL Documentation: `https://dev.mysql.com/doc/`