หลักการ Django DRF ตามแนวคิด Clean Architecture

จุดประสงค์หลัก คือ การแยก Business Logic ออกจาก Framework (Django/DRF) ทำให้โค้ดส่วนของ View ไม่ต้องรู้เรื่องการจัดการฐานข้อมูล (Repository) และโค้ดส่วน Service ไม่ต้องรู้เรื่อง HTTP/JSON (View/Serializer)

1. Model (Data Structure)

ในสถาปัตยกรรมแบบ Clean Architecture, **Model** ที่อยู่ใน models.py ของ Django มีบทบาทที่จำกัดและชัดเจนมาก นั่นคือ การเป็น Entity หรือ โครงสร้างข้อมูล (Data Schema) เท่านั้น และควรมี Logic น้อยที่สุด โดยมองว่า Model คือ **พิมพ์เขียว (Blueprint)** ของข้อมูลที่จะถูกจัดเก็บในฐานข้อมูล

สิ่งที่ควรทำ

- **การกำหนดฟิลด์พื้นฐาน** กำหนดประเภทข้อมูลที่ถูกต้อง เช่น CharField, IntegerField, DateTimeField การใช้ข้อจำกัดในระดับฟิลด์ เช่น max_length, unique=True, null=False ถือเป็น Data Integrity ที่ถูกต้องและควรทำ
- **การกำหนดความสัมพันธ์** ใช้ ForeignKey, ManyToManyField, OneToOneField เพื่อเชื่อมโยงข้อมูลระหว่างตาราง
- **อนุญาต** เมธอดหรือ Property ที่เป็นการ **คำนวณจากฟิลด์ภายในตัวมันเอง** โดยตรงเท่านั้น (เช่น การรวมชื่อ-นามสกุล, การคำนวณอายุจากวันเกิด)

สิ่งที่ไม่ควรทำ

- ห้ามใส่ Logic ที่ซับซ้อนใน Model Methods เด็ดขาด

แหล่งอ้างอิง

การทำความเข้าใจและนำแนวทางจากเอกสารทางการเหล่านี้ไปใช้ จะช่วยสร้างแอปพลิเคชันที่มีคุณภาพสูง, ทำงานได้รวดเร็ว, และบำรุงรักษาได้ง่ายในระยะยาว

1.1 Field Types ทั้งหมดที่มีใน Django - https://docs.djangoproject.com/en/5.0/ref/models/fields/

1.2 ข้อจำกัดให้กับฟิลด์ - https://docs.djangoproject.com/en/5.0/ref/models/fields/#field-options

1.3 วิธีการตรวจสอบความถูกต้องของข้อมูล (เช่น การใช้ validators ในระดับฟิลด์) - https://docs.djangoproject.com/en/5.0/ref/validators/

1.4 การใช้งานและข้อควรพิจารณาสำหรับ Model Relationship Types - https://docs.djangoproject.com/en/5.0/topics/db/models/#relationships

1.5 กลไกสำคัญที่กำหนดว่าจะเกิดอะไรขึ้นกับข้อมูลลูก เมื่อข้อมูลแม่ (Parent) ถูกลบ (เช่น CASCADE, PROTECT, SET_NULL) - https://docs.djangoproject.com/en/5.0/ref/models/fields/#django.db.models.ForeignKey.on_delete

1.6 การเขียน Query ข้อมูลที่เกี่ยวข้อง - https://docs.djangoproject.com/en/5.0/topics/db/queries/#following-relationships-backward

1.7 วิธีการเพิ่มเมธอด (Method) เข้าไปใน Model - https://docs.djangoproject.com/en/5.0/topics/db/models/#adding-custom-methods

1.8 การใช้ Property Decorator ใน Model เพื่อให้เมธอดที่คำนวณค่าสามารถเข้าถึงได้เหมือนฟิลด์ (เช่น การรวมชื่อ-นามสกุล หรือการคำนวณอายุ) - https://docs.python.org/3/library/functions.html#property

1.9 การคำนวณที่ซับซ้อนมาก ที่ต้องการให้ฐานข้อมูลดำเนินการให้โดยตรง - https://docs.djangoproject.com/en/5.0/ref/models/database-functions/

ตัวอย่างเช่น สำหรับ Django DRF Template นี้
มีการสร้างโมเดล CustomUser ในแอป accounts และ CustomUser นี้สืบทอดมาจาก AbstractUser ของ Django และถูกกำหนดให้เป็น User Model หลักของโปรเจกต์ (กำหนดในไฟล์ settings.py ของโปรเจค) สอดคล้องกับแนวคิด Clean Architecture ทำหน้าที่เป็นเพียง **พิมพ์เขียว (Blueprint) ของข้อมูลผู้ใช้งาน** ที่จะถูกเก็บในฐานข้อมูลเท่านั้น ไม่มี Logic ที่ซับซ้อน ดังนั้นจึงสามารถนำไปใช้เชื่อมโยงกับโมเดลอื่น ๆ ได้โดยตรงตามหลักการของ Clean Architecture เช่น Ticket และ Message ที่อาจจำเป็นต้องสร้างในอนาคต

2. Repository (Data Access Layer)
Repository คือ ด่านแรกที่ใช้ในการจัดการข้อมูล เป็นชั้นที่หุ้มการใช้งาน Django ORM ไว้ โดยมีหน้าที่เดียวคือ CRUD (Create, Read, Update, Delete)
การแยกไฟล์
Repository ควรถูกแยกออกมาเป็นไฟล์เฉพาะเพื่อความชัดเจน โดยสร้างไฟล์แยกออกมา เช่น repositories.py หรือสร้างโฟลเดอร์ my_app/repositories/ สำหรับโครงสร้างไฟล์ที่ดีควรมีลักษณะดังนี้

```
my_app/
├──────── models.py
├──────── views.py
├──────── services/
│       ├──────── book_service.py   # Business Logic
├──────── repositories/
│       └──────── book_repository.py  # Data Access Logic (CRUD)
```

บทบาทของ Repository
Repository เป็นเพียง "สะพานเชื่อม" ระหว่าง Service กับฐานข้อมูล โดย Service Layer จะเรียกใช้ Repository และจะไม่เห็นโค้ด ORM เลย ซึ่งหากในอนาคตตัดสินใจเปลี่ยนไปใช้ Cache Layer เช่น Redis ก่อนฐานข้อมูล เราจะสามารถปรับโค้ดใน Repository ให้เช็ค Redis ก่อนได้ โดยที่ Service ไม่ต้องแก้ไขโค้ดเลย หรือถ้าต้องการเปลี่ยนฐานข้อมูลจาก PostgreSQL เป็น MongoDB (เปลี่ยน ORM/ODM) ก็สามารถทำได้โดยแก้ไขแค่ใน Repository Layer เท่านั้น โดยที่ Service Layer ไม่ต้องถูกแก้ไขเลย

ตัวอย่างโค้ดพื้นฐาน (Basic Code Example)
ไฟล์ my_app/repositories/book_repository.py

```
from myapp.models import Book # สมมติว่า myapp.models มี Model ชื่อ Book
```

```
class BookRepository:
    """
    รับผิดชอบในการเข้าถึงและจัดการข้อมูล Book โดยตรง
    ไม่มี Business Logic หรือการตรวจสอบสิทธิ์ในชั้นนี้
    """

    def get_all_books(self):
        """ดึงหนังสือทั้งหมด"""
        return Book.objects.all()

    def get_book_by_id(self, book_id):
        """ดึงหนังสือตาม ID หรือคืนค่า None ถ้าไม่พบ"""
        try:
            return Book.objects.get(pk=book_id)
        except Book.DoesNotExist:
            return None

    def create_book(self, data):
        """สร้างรายการหนังสือใหม่จาก dict ของข้อมูล"""
        # ใช้ **data เพื่อ unpack ข้อมูลที่ได้รับจาก Service
        return Book.objects.create(**data)

    def update_book(self, book_instance, update_data):
        """อัปเดตข้อมูลของ Book instance ที่มีอยู่แล้ว"""
        for key, value in update_data.items():
            setattr(book_instance, key, value)
        book_instance.save()
        return book_instance
```

ไฟล์ my_app/services/book_service.py

```
# นำเข้า Repository
from myapp.repositories.book_repository import BookRepository

# สร้าง Instance ของ Repository
book_repo = BookRepository()

class BookService:
    """
    รับผิดชอบ Business Logic เช่น การตรวจสอบความถูกต้อง, การคำนวณราคา,
```

```
    และการจัดการ Transaction
    """

    def get_book_list_for_display(self):
        """
        Business Logic: ดึงรายการหนังสือ และอาจมีการกรองหรือประมวลผลเพิ่มเติม
        """
        # Service เรียก Repository - ไม่เห็นโค้ด ORM
        books = book_repo.get_all_books().filter(is_published=True)

        # ... อาจมี Logic อื่นๆ ตรงนี้ ...

        return books

    def create_new_book(self, book_data, user):
        """
        Business Logic: ตรวจสอบสิทธิ์ผู้ใช้ก่อนสร้าง
        """
        if not user.is_staff:
            raise PermissionError("User not authorized to create books.")

        # Service เรียก Repository - ส่งข้อมูลดิบไปให้จัดการ
        new_book = book_repo.create_book(book_data)

        return new_book
```

3. Service (Business Logic Layer)

Service Layer เป็นชั้นที่อยู่ระหว่าง View (Presentation Layer) และ Repository (Data Access Layer) หน้าที่หลัก คือ การดำเนินการตาม "กฎทางธุรกิจ" ที่ซับซ้อน เช่น การคำนวณ, การตรวจสอบสิทธิ์, และการจัดการ Transaction เป็นต้น เป็นที่อยู่ของ Business Logic ทั้งหมด โดยเป็นชั้นที่รับคำสั่งจาก View และประสานงานกับ Repository

การแยกไฟล์ (File Structure)

Service Logic ควรถูกแยกออกมาในโฟลเดอร์เฉพาะ เพื่อให้โค้ดส่วน View ดูสะอาด และง่ายต่อการค้นหา Logic ทางธุรกิจ

```
my_app/
  ├──────── models.py
  ├──────── views.py         # ทำหน้าที่เพียงรับ Request และส่งต่อให้ Service
  ├──────── repositories/
  │         └──────── book_repository.py
  └──────── services/
      ├──────── book_service.py    # Business Logic (หนังสือ)
      └──────── order_service.py   # Business Logic (คำสั่งซื้อ)
```

การพึ่งพา (Dependency Injection)

Service Layer ช่วยให้สามารถประสานงานกับ Repository สองตัวเพื่อทำ Transaction ทางธุรกิจที่ต้องการได้โดยง่าย

ไฟล์ my_app/services/order_service.py

```
class OrderService:
    def __init__(self, order_repo, inventory_repo):
        self.order_repo = order_repo
        self.inventory_repo = inventory_repo

    def place_order(self, cart_data, user):
        # 1. Logic ตรวจสอบสถานะผู้ใช้, คำนวณภาษี/ส่วนลด

        # 2. Orchestration (ประสานงาน) และ Transaction Management
        with transaction.atomic():
            # Service เรียก Inventory Repository เพื่อลด Stock
            is_successful = self.inventory_repo.reduce_stock(cart_data)

            if not is_successful:
                raise Exception("Out of stock.")

            # Service เรียก Order Repository เพื่อสร้างคำสั่งซื้อ
            order = self.order_repo.create_order(cart_data, user)

        # 3. Logic ส่งอีเมลยืนยัน

        return order
```

4. Serializer และ View (API/Presentation Layer)

4.1 Serializer ใน Django Rest Framework (DRF)

มีบทบาทที่ชัดเจนและจำกัด ไม่ควรมี Business Logic อยู่ในชั้นนี้ เช่น รับผิดชอบในการตรวจสอบความถูกต้องของข้อมูล (Format Validation) ที่มาจาก Client หรือแปลง Model Instance (Python Objects) ให้เป็นข้อมูลที่สามารถส่งออกไปภายนอกได้

สิ่งที่ควรทำ

- **ModelSerializer** ใช้เพื่อเชื่อมโยงกับ Django Model โดยอัตโนมัติ ทำให้โค้ดกระชับและลดความซ้ำซ้อน
- กำหนด fields ที่ต้องการให้แสดงออก และ read_only_fields สำหรับฟิลด์ที่ไม่ต้องการให้ Client แก้ไข

สิ่งที่ไม่ควรทำ

- Serializer ห้ามเรียกใช้ Repository หรือทำการคำนวณที่ซับซ้อน เช่น การตรวจสอบสิทธิ์, การจัดการคลังสินค้า, หรือการคำนวณราคาสุทธิที่ซับซ้อน เป็นต้น การดำเนินการเหล่านี้เป็นของ **Service Layer**

4.2. View Layer (APIView/ViewSet)

View Layer คือ จุดรวมที่รับผิดชอบการจัดการ HTTP Request/Response และการส่งต่อคำสั่งไปยัง Service Layer ในแอป เช่น ใช้ DRF Permissions เพื่อควบคุมว่าใครสามารถเข้าถึง View นี้ได้ ก่อนส่งต่อให้ Service เป็นต้น

View ที่ดีควรทำตาม 5 ขั้นตอนนี้เสมอ

- **รับ Request** โดยรับข้อมูลจาก request.data
- **ตรวจสอบข้อมูลขาเข้า** โดยใช้ Serializer ในการ validate() ข้อมูล
- **เรียกใช้ Service Layer** โดยส่งต่อข้อมูลที่ผ่านการตรวจสอบแล้วไปยัง Service Layer เพื่อดำเนินการ Business Logic
- **จัดรูปแบบข้อมูลขาออก** โดยใช้ Serializer เพื่อแปลงผลลัพธ์จาก Service ให้เป็น JSON
- **ส่ง Response** โดยคืน HTTP Response Status เช่น 200, 201

ตัวอย่างโค้ดพื้นฐาน

ไฟล์ Serializer

```
from rest_framework import serializers
from myapp.models import Book

class BookSerializer(serializers.ModelSerializer):
    # ฟิลด์ที่คำนวณภายในตัว Model เอง สามารถแสดงผลเป็น read-only ได้
    full_title = serializers.ReadOnlyField()

    class Meta:
        model = Book
        fields = [
```

```
            'id',
            'title',
            'price',
            'publisher_id',
            'full_title' # ฟิลด์ที่คำนวณ
        ]
        read_only_fields = ['id', 'full_title']

    # Serializer Validation ใช้เพื่อตรวจสอบความถูกต้องของรูปแบบข้อมูลเท่านั้น
    def validate_price(self, value):
        if value < 0:
            raise serializers.ValidationError("Price cannot be negative.")
        return value
```

ไฟล์ View Layer

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status

# นำเข้า Service และ Serializer
from myapp.serializers.book_serializer import BookSerializer
from myapp.services.book_service import BookService
from myapp.repositories.book_repository import BookRepository

# สร้าง Instance ของ Service และ Repository
book_repo = BookRepository()
book_service = BookService(book_repo=book_repo)

class BookCreateAPIView(APIView):
    # สามารถกำหนด Permission ที่นี่
    # permission_classes = [permissions.IsAuthenticated]

    def post(self, request):
        # 1. รับ Request และ 2. ใช้ Serializer ตรวจสอบข้อมูลขาเข้า
        serializer = BookSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)

        try:
            # 3. เรียกใช้ Service Layer เพื่อดำเนินการ Business Logic
```

```
        validated_data = serializer.validated_data
        user = request.user

        # Service รับผิดชอบในการสร้างและจัดการ Logic ทั้งหมด
        book_instance = book_service.create_new_book(validated_data, user)

        # 4. ใช้ Serializer จัดรูปแบบข้อมูลขาออก
        response_serializer = BookSerializer(book_instance)

        # 5. ส่ง Response
        return Response(response_serializer.data, status=status.HTTP_201_CREATED)

    except PermissionError as e:
        return Response({"detail": str(e)}, status=status.HTTP_403_FORBIDDEN)
    except Exception as e:
        return Response({"detail": "An unexpected error occurred."}, status=status.HTTP_400_BAD_REQUEST)
```

การปรับใช้แนวคิด Clean Code (Model-Repository-Service) เข้ากับโครงสร้าง Modular Monolith ใน Django DRF

1. หลักการของ Modular Monolith ใน Django

Modular Monolith คือ การจัดโครงสร้างโค้ดในโปรเจคให้เป็นโมดูลอิสระที่ทำงานร่วมกัน โดยมีหลักการสำคัญ คือ **หนึ่ง App ต่อหนึ่ง Domain** โดยแต่ละ Django App ควรเป็นตัวแทนของโดเมนทางธุรกิจที่มีความรับผิดชอบเดียว

ตัวอย่างโครงสร้าง

```
project_root/
    users/ (Domain: การจัดการผู้ใช้)
    products/ (Domain: การจัดการสินค้า)
    orders/ (Domain: การจัดการคำสั่งซื้อ)
```

2. โครงสร้างภายในแต่ละแอป

ภายในแต่ละ Domain App ควรมีโครงสร้างไฟล์ที่แยก Layer อย่างชัดเจน แทนที่จะรวมทุกอย่างไว้ในไฟล์เดียว

โครงสร้างไฟล์ที่แนะนำ

```
/users
  ├──────── models.py
  ├──────── serializers.py
  ├──────── views.py          <-- Presentation Layer (เรียก Service)
  ├──────── services/
```

```
│   └──── user_service.py     <-- Business Logic Layer
├──── repositories/
│   └──── user_repository.py  <-- Data Access Layer (เรียก ORM/Model)
└──── tests/
    ├──── test_user_service.py
    └──── test_user_repository.py
```

สิ่งที่ควรทำ

No Cross-Domain ORM Access เช่น ห้าม UserService หรือ UserRepository import หรือเข้าถึง ProductModel หรือ ProductRepository เด็ดขาด เพราะนั่นคือการทำลาย Boundaries ของโดเมน

3. การสื่อสารระหว่างโดเมน (Inter-Module Communication)

จุดที่สำคัญที่สุดในการรักษา Clean Code ใน Modular Monolith เพราะการ import ข้ามโดเมนจะทำให้เกิด Tight Coupling ควรสื่อสารโดยยึดหลักการที่เรียกว่า "Service-to-Service Delegation"

3.1 การเรียกใช้แบบตรง (Direct Service Delegation)

เมื่อ Service ใน App A ต้องการข้อมูลหรือ Action จาก App B ให้เรียกผ่าน Service ของ App B เท่านั้น

ไฟล์ orders/services/order_service.py

```
# 1. Import Service จากโดเมนอื่น
from products.services.product_service import ProductService

class OrderService:
    def __init__(self, product_service: ProductService):
        self.product_service = product_service

    def create_order(self, order_data):
        # ... Logic สร้าง Order ...

        # 2. เรียก Business Logic ของโดเมนอื่น
        self.product_service.reduce_stock(order_data['product_id'], order_data['quantity'])

        # ... Logic บันทึก Order ...
        return order
```

3.2 การสื่อสารแบบหลวม (Events/Signals)

สำหรับ Action ที่ไม่จำเป็นต้องตอบกลับทันที หรือเป็นการแจ้งให้โดเมนอื่นรับทราบและทำบางสิ่ง ให้ใช้ Django Signals หรือระบบ Event-Driven

การรวม Stateless Architecture (Redis) และ Asynchronous Tasks (Celery) เข้ากับโครงสร้าง Modular Monolith แบบ Clean Code

เป็นขั้นตอนสำคัญในการสร้างแอปพลิเคชันที่ Scalable และมี Performance สูงขึ้น

1. Stateless Architecture และ Redis

หลักการของ **Stateless Architecture** คือ การออกแบบให้เซิร์ฟเวอร์ (Django Process) ไม่เก็บข้อมูลสถานะของผู้ใช้ไว้ระหว่าง Request ต่าง ๆ เช่น Session, Cart Status เป็นต้น ซึ่งจะทำให้สามารถเพิ่มหรือลดจำนวนเซิร์ฟเวอร์ได้อย่างง่ายดาย

1.1 บทบาทของ Redis

- **Cache (แคช)** ใช้เก็บผลลัพธ์ของข้อมูลที่มีการคำนวณซ้ำบ่อย ๆ เพื่อลดการเข้าถึงฐานข้อมูลหลัก
- **Message Broker (ตัวกลาง)** ใช้เป็นช่องทางสื่อสารระหว่าง Django Application (Publisher) กับ Celery Worker (Consumer) ซึ่งเป็น Process แยกต่างหาก

1.2 การใช้งาน Cache

การเพิ่ม Caching ควรทำใน Layer ที่เหมาะสมที่สุดเพื่อรักษาความ Clean ได้แก่ Repository Layer
เนื่องจาก Repository มีหน้าที่ในการเข้าถึงข้อมูลเท่านั้น การจัดการ Cache จึงถือเป็นความรับผิดชอบของ Repository Layer เพื่อให้มั่นใจว่า Service Layer จะยังคง "Clean" โดยไม่ต้องสนใจว่าข้อมูลนั้นถูกดึงมาจาก Cache หรือ Database

2. การผนวก Celery Worker (Asynchronous Tasks)

Celery คือ เครื่องมือสำหรับการประมวลผลแบบอะซิงโครนัส (Asynchronous Processing) ซึ่งช่วยในการย้ายงานที่ใช้เวลานาน เช่น การส่งอีเมล, การสร้างรายงาน, การประมวลผลภาพ เป็นต้น ออกจากวงจร HTTP Request ทำให้ API ตอบสนองได้เร็วขึ้น

2.1 การจัดโครงสร้าง Celery Task ใน Modular Monolith

Tasks ควรอยู่ภายในโดเมนที่เกี่ยวข้อง เพื่อรักษาหลักการ Encapsulation ของ Modular Monolith

- **ที่ตั้ง:** สร้างไฟล์ tasks.py ภายในแต่ละ Domain App เช่น users/tasks.py, orders/tasks.py เป็นต้น

2.2 Celery Task ต้องเรียกใช้ Service Layer เท่านั้น

Celery Task ต้องทำหน้าที่เป็นเพียง Wrapper (ตัวห่อหุ้ม) เท่านั้น ห้ามใส่ Business Logic หรือเรียก ORM/Repository โดยตรง

3. ภาพรวมการไหลของข้อมูล (The Clean Flow)

ภาพรวมของการทำงานข้าม Layer และข้าม Process ในสถาปัตยกรรมที่รวมกันนี้

- **View (API Layer)** รับ HTTP Request เช่น POST /users/report/ และเรียก Service
- **Service (Business Logic Layer)** ดำเนินการ Logic ที่จำเป็น เช่น การตรวจสอบสิทธิ์ เป็นต้น โดยเมื่อพบ Task ที่ต้องใช้เวลานาน เช่น การสร้างรายงาน จะเรียก **Celery Task** แบบ Asynchronous
- **Celery Task (Wrapper)** ส่ง Job/Message ไปยัง Redis Broker
- **Redis Broker** จัดคิว (Queue) งานที่ต้องทำ

- **Celery Worker (Process แยก)** ดึงงานจาก Queue มาทำ โดยการ **เรียก Service Layer Method**
- **Service Layer** รัน Business Logic อย่างสมบูรณ์
- **Repository Layer:** จัดการกับ Database หรือ Redis Cache (ตาม Logic ที่กำหนดไว้)

สิ่งที่ควรทำ

Statelessness ใช้ JWT/Token สำหรับ User State และใช้ Redis สำหรับ Caching และ Broker
Decoupling Celery Task ห้ามบรรจุ Logic แต่ต้องเรียก Service Layer เท่านั้น (Service Contract)
Encapsulation ทุก Task ที่เกี่ยวข้องกับโดเมนใดต้องอยู่ภายใน App ของโดเมนนั้น

การรักษาความปลอดภัยและการยืนยันตัวตน

ในสถาปัตยกรรม Clean Code การจัดการการยืนยันตัวตน (Authentication) และการอนุญาตสิทธิ์ (Authorization) จะถูกแบ่งความรับผิดชอบอย่างชัดเจนระหว่าง View Layer และ Service Layer

A. JWT (JSON Web Tokens) และ Stateless Architecture

หลักการ คือ ควรใช้ JWT แทน Session-based Authentication เพื่อให้ API เป็นแบบ Stateless (ไร้สถานะ) ซึ่งสอดคล้องกับหลักการ Scalable Architecture

Authentication (การยืนยันตัวตน) กระบวนการตรวจสอบว่าผู้ใช้ เป็นใคร ควรเกิดขึ้นที่ View Layer โดยเมื่อ View ได้รับ Token, DRF จะถอดรหัสและกำหนด request.user ให้เป็น User Object ที่ถูกต้อง การดำเนินการนี้เกิดขึ้นก่อน ที่ Request จะเข้าสู่ Business Logic

B. Authorization (การอนุญาตสิทธิ์)

การอนุญาตสิทธิ์ (Authorization) ถูกแบ่งออกเป็น 2 ระดับ คือ ระดับ API Endpoint และระดับ Business Logic:

1. การจัดการสิทธิ์ใน View Layer (API Endpoint Authorization)

ใช้ rest_framework.permissions เพื่อควบคุมว่า User Object ที่ยืนยันตัวตนแล้ว มีสิทธิ์เข้าถึง API Endpoint นั้นหรือไม่ โดยประเภทของ Permission ได้แก่

- IsAuthenticated ผู้ใช้ต้องเข้าสู่ระบบ
- IsAdminUser ผู้ใช้ต้องเป็นผู้ดูแล
- Custom Permissions สร้างคลาสเองเพื่อตรวจสอบสิทธิ์พื้นฐานตาม Role (เช่นกรณีของ Template นี้)

ข้อดี เป็นการป้องกันด่านแรกที่รวดเร็ว และทำให้ Service Layer ไม่ต้องกังวลเรื่องการตรวจสอบสิทธิ์พื้นฐาน

2. การจัดการสิทธิ์ใน Service Layer (Business Logic Authorization)

หากมีการตรวจสอบสิทธิ์ที่ซับซ้อนตาม Business Logic หรือเกี่ยวข้องกับความสัมพันธ์ของข้อมูล (Object-level Permission) ต้องมี Logic นี้อยู่ใน Service Layer เช่น สถานการณ์ที่ "ผู้ใช้ A สามารถแก้ไขได้เฉพาะรายการ Product ที่ตนเองสร้างเท่านั้น" เป็นต้น