Djagno 객체지향

장고에서 객체지향을 이해한다는 것은 결국 경계와 책임을 정하는 일이다. 파이썬은 엄격한 접근 제한자 대신 관례와 프로토콜에 기대며, 덕 타이핑을 통해 "이 메서드를 지원하면 그 역할을 수행할 수 있다"라는 느슨한 합의를 가능하게 한다. 장고는 이러한 유연함을 전제로 모델/쿼리셋/CBV/폼/시리얼라이저/미들웨어/스토리지 같은 구성요소에 명확한 확장 포인트를 놓았다. 프레임워크의 의도는 "핵심 알고리즘의 골격은 유지하되, 필요한 지점만 얇게 바꿔라"이며, 그 수단이 **템플릿 메서드, 믹스인, 전략(Strategy), 어댑터(Adapter)**다. 초급 단계에서 가장 중요한 태도는 "어디에 무엇을 숨기고, 어디에서 무엇을 드러낼지"를 일관되게 결정하는 습관을 들이는 것이다.

먼저 모델을 도메인 객체로 대한다는 감각부터 세우자. 모델은 테이블의 얇은 투영이 아니라, 상태와 불변조건을 품은 단위다. "주문은 결제 승인 전 확정될 수 없다", "게시물은 공개 시점이 현재보다 과거일 수 없다" 같은 규칙은 템플릿이나 뷰가 아니라 모델 메서드 근처에서 검사되어야 한다. 실무에서는 뷰에서 값만 검증하고 저장한 뒤, 별도의 헬퍼로 비즈니스 규칙을 흩뿌리는 패턴이 자주 보인다. 이렇게 흐르면 규칙이 중복되고 테스트 범위가 뷰 단위로만 엉성해진다. 반대로, 모델이 규칙을 품게 하면 어느 경로(REST API, 관리자 페이지, 배치 작업)로 호출하든 일관된 결과가 나온다. 저장 직전 검증은 무턱대고 save()를 오버라이드하기보다, 유효성의 책임과 시점을 분리해 full_clean()을 의식적으로 호출하거나, 명시적 도메인 메서드 내부에서 검사를 끝낸 후 저장하도록 흐름을 정리하는 편이 안전하다. 신호(pre_save, post_save)는 강력하지만 흐름을 숨긴다. 전역 전파라는 특성상 추적성이 떨어지므로, 핵심 규칙보다는 부수효과(예: 알림 큐 발행) 정도에만 제한하는 문화가 건강하다.

모델 상속은 중복 제거와 다형성 도입의 기본 도구다. 추상 기반 클래스로 공통 필드(생성/수정 시각, 작성자, 소프트 삭제 플래그 등)와 반복 동작을 합치면 코드량이 줄고 일관성이 생긴다. 반면 다중 테이블 상속은 신중해야 한다. ORM 레벨에서의 다형성은 우아해 보이지만 조인이 깊어지면서 실행 계획이 복잡해지고, 대량 조회에서 성능 비용이 누적된다. 필요하다면 프록시 모델로 표현만 바꾸거나, 상속 대신 **조합(Composition)**으로 협력을 설계해도 된다. 한편 모델이 점점 비대해져 '신(God) 모델'이 되기 시작하면, 상태 전이는 모델에 둔 채 유스케이스 조립을 맡을 도메인 서비스 클래스를 분리해 책임을 가볍게 만들자. 서비스는 순수 파이썬으로 작성하고, 메서드 단위에 transaction.atomic()을 두어 경계를 또렷하게 만든다. 이때 매니저는 최소 질의를, 모델은 상태 규칙을, 서비스는 시나리오를 맡는 식의 분업이 유지되면 결합도가 자연스레 내려간다.

장고의 **쿼리셋(QuerySet)**은 객체지향 빌더다. 체이닝으로 조건을 쌓고, 게으른 평가로 필요할 때만 SQL을 보낸다. 이 설계는 템플릿 메서드와 불변(에 가깝게 느껴지는) 작은 객체들을 연결해 만든다. 초급자가 가장 자주 범하는 실수는 루프 안에서 관계 속성을 접근해 N+1 쿼리를 만든다는 점이다. 객체지향 관점에서 보면 "속성 접근"처럼 보이는 행위가 사실은 "DB 왕복"을 트리거한다. 이를 예방하는 사고법은 단순하다. 접근 전에 가져온다. 외래키·일대일 같은 단일 관계는 select_related로 조인해 한 번에 묶고, 일대다·다대다 관계는 prefetch_related로 별도 쿼리 한 번 더 날려 파이썬에서 연결한다. 이렇게 하면 템플릿이나 뷰에서는 편안한 다형성 인터페이스만 써도 쿼리 수가 상수로 유지된다. 필요한 컬럼만 얇게 나르는 only/defer, 딕셔너리 형태로 건네는 values/values_list, 메모리 폭주를 피하는 iterator 같은 투영과 스트리밍 도구는 병목이 확인된 후 단계적으로 도입하는 게 현실적이다. ORM은 SQL을 숨기는 도구가 아니라 써야 할 SQL을 정확히 쓰게 만드는 도구라는 감각을 잃지 말자.

클래스 기반 뷰(CBV)는 장고의 객체지향이 특히 또렷하다. dispatch → http_method_not_allowed → get/post → get_queryset → get_context_data로 이어지는 후크 체인은 템플릿 메서드의 전형이다. 프레임워크가 상위 알고리즘의 골격을 제공하고, 사용자는 필요한 고리만 얇게 오버라이드한다. 이 장점의 이면에는 믹스인 조합의 **MRO(메서드 해결 순서)**가 있다. 믹스인이 super를 건너뛰면 체인이 끊기고, 의도치 않은 동작이 생긴다. 믹스인은 "작고 한정된 책임 하나만 수행한다"는 규율 아래, 항상 super()를 호출하는 습관을 들이자. 권한 검사, 로그인 요구, 공통 필터링, 페이지네이션 같은 가로 단 요구는 믹스인으로, 도메인 특화 동작은 뷰 내부에서 명시적으로 구현하면 조합의 복잡도를 낮출 수 있다. 제네릭 뷰(List/Detail/Create/Update/Delete)를 우선 익히고, 팀에서 반복되는 패턴을 두세 개의 안정적인 믹스인으로만 표준화해도 체감 유지보수성이 크게 오른다.

DRF까지 확장하면 객체지향의 조합력이 더 강해진다. 시리얼라이저는 데이터의 외부 표현을 담당하는 전략 객체이고, 뷰셋은 HTTP 메서드와 유스케이스를 묶는 컨트롤러 역할을 한다. 하나의 도메인 객체를 여러 표현으로 내보낼 때, 서로 다른 시리얼라이저를 주입해 다형성을 얻는다. 권한(permissions), 인증(authentication), 스로틀(throttling), 필터/filter backends는 그대로 전략 패턴이며, 설정·속성으로 교체 가능하므로 개방-폐쇄 원칙에 부합한다. 다만 실무에서는 시리얼라이저가 검증과 저장, 부수효과까지 떠안아 비대해지기 쉽다. 이럴수록 "표현은 시리얼라이저, 규칙은 모델/서비스" 원칙으로 되돌아가자. validate_*(field)나 validate에서는 형식적·상호관계 검증만 하고, 도메인 룰은 모델/서비스 호출로 위임하라. API 버전이 갈라지는 시점에는 기존 뷰셋·시리얼라이저에 조건 분기를 쌓기보다, 얇은 어댑터를 두거나 독립 클래스로 복제해 병행 운영하는 편이 변경의 파급을 통제하기 쉽다.

객체지향의 핵심인 캡슐화는 파이썬에서 관례와 모듈 경계로 실천된다. 중요한 것은 "무엇을 숨길지"를 정하는 일이다. 외부 결제, SMS, 서드파티 API 같은 불안정한 의존성은 뷰에서 직접 호출하지 말고, 포트/어댑터 구조로 감싸라. 인터페이스는 "승인/취소/정산" 같은 의미 있는 동사만 드러내고, 구체 구현은 설정에서 문자열 경로로 지정해 import_string으로 로드하면 운영·테스트 환경 전환도 쉽다. 거창한 DI 컨테이너 없이도, "생성 책임을 모아둔 작은 팩토리"만 있으면 팀 규모에서 충분히 유연해진다. 이때 타입 힌트와 typing.Protocol을 아끼지 말라. 덕 타이핑의 자유 위에 도구 친화성을 더해 준다.

상속과 조합 사이에서 흔들릴 때는 변경 이유를 기준으로 선택하라. 한 클래스가 둘 이상의 이유로 변한다면 분리 신호다. 모델 메서드로 모든 유스케이스를 흡수하면 테스트가 어려워지고, 반대로 서비스에 모든 것을 몰면 ORM의 이점을 잃는다. "상태 제약과 불변은 모델, 시나리오 조립은 서비스"라는 분업을 중심축으로 잡되, 조회/리포팅처럼 읽기가 본질인 영역은 아예 "읽기용 서비스(쿼리 서비스)"로 분리해 성능 최적화를 전제하고 출발하는 편이 낫다. 이벤트 처리도 비슷하다. 신호로 전역 브로드캐스트하기 전에, 명시적 이벤트 퍼블리셔를 도메인에 두고, 나중에 필요하면 메시지 큐로 바꿀 수 있게 경계를 설계한다. 데이터 일관성이 중요한 곳은 아웃박스 패턴처럼 저장과 발행을 한 트랜잭션에 묶어두는 구조가 실무에서 안전하다.

트랜잭션과 동시성은 객체지향의 경계 설계와 직결된다. 유스케이스 단위로 transaction.atomic()을 묶고, 경쟁 조건이 우려되는 구간에서는 select_for_update()로 행 잠금을 걸어 데이터 경합을 줄인다. 증감 연산은 F() 표현으로 원자적 업데이트를 쓰면 조회-변경-저장 사이 레이스를 줄인다. 멱등성이 필요한 작업(결제 승인, 포인트 적립 등)은 요청에 멱등 키를 부여하고, 서버는 처리 이력을 별도 테이블이나 캐시에 기록하여 중복 반영을 막는다. 이런 규율은 코드 가독성 이상의 운영 안전성을 가져온다. 반대로 비동기 후행 작업은 Celery 등으로 떼어내되, 함수 인터페이스에서 입력은 순수 데이터, 부수효과는 내부에서만 발생하게 만들어 재시도에도 안전하도록 구성한다. 실패 시 보상(취소/환불)이 필요한 시나리오는 **사가(Saga)**처럼 보상 트랜잭션을 명시적으로 모델링해야 한다.

성능은 객체 사이의 협력을 어떻게 설계하느냐의 결과다. 고가용성을 이유로 무턱대고 캐시를 도입하기보다, 먼저 프로파일링으로 병목을 확인하라. 장고 디버그 툴바나 쿼리 로깅만 잘 켜도 어느 뷰에서 몇 개의 쿼리가 나가고, 어떤 쿼리가 느린지 금방 보인다. 병목이 확인되면, 관계 순회를 줄이는 select_related/prefetch_related, 필요 컬럼만 투영하는 values/only/defer, 대량 생성·갱신에는 bulk_* 계열, 카운트·집계는 DB에게 넘기는 식으로 역할을 바르게 배분한다. 한편 캐시는 경계 객체로 두어 도메인 모델을 오염시키지 말고, Cache-Aside 패턴으로 "먼저 캐시 조회 → 미스면 DB 조회 → 캐시에 저장" 흐름을 만들되, 키 설계와 만료/무효화 전략을 사전에 정한다. 읽기 일관성이 엄격히 필요 없는 데이터(랭킹, 추천 등)는 TTL을 짧게 두고 재생성하는 구조가 단순하고 안전하다. 반대로 권한·설정 같은 민감한 데이터는 캐시를 아예 피하거나, 버전 키를 붙여 정확성을 우선한다.

요청-응답 생명주기를 객체지향으로 읽어도 유익하다. 서버(WSGI/ASGI)가 요청을 받으면 HttpRequest가 만들어지고, 미들웨어 체인이 전처리를 수행한 뒤 URL 리졸버가 뷰 객체를 찾는다. 뷰는 도메인 서비스를 호출하고, 쿼리셋으로 모델을 가져오며, 템플릿/시리얼라이저가 외부 표현으로 가공한다. 각 단계는 확장 포인트이자 절제 포인트다. 전역적인 요구(추적 ID 주입, 공통 헤더, 로깅 컨텍스트)는 미들웨어에서, 엔드포인트별 요구는 뷰·시리얼라이저에서, 도메인 규칙은 모델/서비스에서 맡는다. 이렇게 경계를 분명히 하면, 문제를 디버깅할 때도 "어느 층에서 깨졌는지"를 빠르게 좁힐 수 있다. ASGI 환경에서의 비동기 뷰는 지연이 큰 I/O를 숨기는 데 유리하지만, CPU 바운드 작업은 여전히 큐로 분리하거나 C 확장/멀티프로세싱을 고려해야 한다. 객체지향적으로는 "동일한 인터페이스를 가진 동기/비동기 구현"을 바꿔 끼울 수 있게 설계해 두는 것이 장기적으로 편하다.

API 설계는 객체지향 모델링과 나란히 간다. 리소스 경로는 명사로, 상태 전이는 HTTP 메서드와 상태 코드로 표현한다. 멱등이 필요한 곳은 PUT/PATCH를 쓰고, 생성은 POST에 묶되 중복 제출을 고려해 클라이언트·서버 모두에서 멱등 키를 다룬다. 표현 계층과 도메인 계층의 분리를 위해, 내부 모델 그 자체를 외부로 내보내지 말고 **표현 객체(시리얼라이저)**가 역할을 맡게 하자. 오류는 도메인 예외 → HTTP 응답 매핑 규칙을 정해 일관된 형식으로 내보내면, 클라이언트가 회복 전략을 세우기 쉬워진다. 버저닝은 "새 버전은 새로운 타입"이라는 관점으로 접근해, 기존 클래스에 조건 분기를 늘리기보다 독립 클래스로 분기하는 편이 예측 가능성을 높인다. 무엇보다 리소스 경계 = 집합체(aggregate) 경계라는 DDD 감각을 유지하면, 일관성·락 범위·캐시 키 설계가 자연스럽게 정렬된다.

테스트는 설계를 거울처럼 비춘다. 테스트하기 쉬운 코드가 곧 좋은 객체지향이다. 모델은 팩토리로 손쉽게 만들 수 있고, 서비스는 트랜잭션 경계를 기준으로 시나리오 테스트를 구성하되 성공/실패/보상 흐름을 명확히 검증한다. 외부 연동은 포트/어댑터 인터페이스를 **더블(스텁/스파이)**로 교체해 단위 테스트를 빠르게 돌리고, 실제 구현은 통합 테스트로 최소한만 담보한다. DRF에서는 API 테스트에서 쿼리 수 단언도 유용하다. N+1이 재발하지 않도록 특정 요청이 상수 개 쿼리만 나가야 한다고 못 박아두면, 리팩토링 과정에서 퍼포먼스 회귀를 초기에 잡을 수 있다. 테스트가 설계를 이끈다는 말은 과장이 아니다. "테스트를 쉽게 쓰기 위해 인터페이스를 단순화하고 의존성을 분리한다"는 압력이, 결국 팀이 유지할 수 있는 코드 구조를 만든다.

실무에는 늘 반(反)패턴이 있다. 첫째, 신호 남용: 여러 앱에 산재한 리시버가 전역 상태를 바꾸면 추적성이 무너진다. 명시적 호출로 바꾸고, 꼭 필요할 때만 신호를 쓰자. 둘째, 비대한 시리얼라이저: 검증·저장·부수효과가 한데 섞이면 수정이 무섭다. 표현과 규칙을 분리하라. 셋째, God 모델: 모든 유스케이스가 모델 메서드로 몰리면 협력자가 사라진다. 서비스로 유스케이스를 끌어올리되, 모델의 불변조건은 남겨라. 넷째, ORM 과신: 복잡한 집계·리포팅을 억지로 ORM으로 짜다 성능과 가독성을 동시에 잃는다. 뚜렷한 이유가 있으면 Raw SQL/뷰/머티리얼라이즈드 뷰를 쓰고, 코드를 얇게 감싸 유지보수성을 확보하라. 다섯째, 믹스인 과잉: 두세 개로 충분할 것을 다섯·여섯 개로 쌓아 올리면 MRO 지옥이 된다. 믹스인은 적을수록 좋다. 여섯째, 마이그레이션 경시: 스키마 진화는 제품의 생명력이다. 뒤로 호환되는 변경을 선호하고, 롤아웃→더

블라이트→스위치→클린업의 이행 절차를 지키면 다운타임 없는 전환이 가능하다.

이 모든 논의를 하나의 사례로 묶어 보자. "주문-결제" 도메인을 장고로 모델링한다고 가정한다. 주문은 NEW → PENDING_PAYMENT → PAID → FULFILLED/FAILED와 같은 상태 머신을 가진다. 상태 전이는 Order.confirm(), Order.mark_paid(), Order.fail() 같은 명시적 메서드로만 허용하고, 각 메서드는 현재 상태를 검사해 불법 전이를 막는다. 결제 어댑터는 PaymentPort 프로토콜을 따르며, 구현은 설정에 따라 주입한다. 결제 승인 유스케이스는 서비스 pay(order_id, idempotency_key)가 맡아 atomic() 블록에서 select_for_update()로 행을 잠그고, 외부 승인 성공 시 Order.mark_paid()를 호출한다. 멱등 키는 "주문ID+클라이언트키" 조합으로 별도 테이블에 기록해 중복 호출을 무해화한다. DRF 뷰셋은 주문 CRUD를 제공하되, pay는 행위 엔드포인트로 POST /orders/{id}/pay에 배치한다. 응답 표현은 "외부 표현"을 위한 시리얼라이저가 맡고, 내부 모델은 외부에 노출하지 않는다. 목록 조회는 select_related('customer')와 prefetch_related('lines')로 상수 쿼리를 보장하고, 핫한 목록은 캐시 키 orders:list:{customer}:{page}로 Cache-Aside를 적용한다. 테스트는 상태 전이의 합법·불법 케이스, 멱등성, 동시 결제 레이스, 캐시 무효화, 그리고 쿼리 수 단언까지 커버한다. 이 한 흐름만 제대로 설계·구현·검증해도, 장고에서의 객체지향이 무엇을 의미하는지 손끝으로 체득하게 된다.

마지막으로, **관찰성(Observability)**을 객체지향의 연장선에서 본다. 로깅은 문자열을 찍는 행위가 아니라, 도메인 이벤트의 기록이다. 컨텍스트(요청 ID, 사용자 ID, 상관관계 ID)를 미들웨어에서 주입하고, 도메인 서비스는 의미 있는 이벤트(주문 확정, 결제 승인, 재고 차감)를 구조화된 로그로 남긴다. 예외는 타입으로 소통한다. 도메인 예외는 계층적으로 정의하고, DRF의 예외 핸들러에서 HTTP 응답으로 공정하게 매핑한다. 트레이싱은 뷰→서비스→어댑터까지 스팬을 흘려보낼 수 있게 인터페이스에 컨텍스트를 가볍게 통과시키면 된다. 이런 설계는 장애 원인 분석 시간을 줄이고, 팀이 자신 있게 변경할 수 있게 만든다.

정리하면, 장고에서의 객체지향은 패턴 암기가 아니다. 변화와 제약 속에서 "무엇을 숨기고 무엇을 드러낼지", "어디서 결정을 내리고 어디서 위임할지", "어떤 곳을 확장 가능하게 두고 어떤 곳을 닫아둘지"를 반복해서 선택하는 기술이다. 모델은 상태와 규칙을, 서비스는 시나리오를, 뷰/시리얼라이저는 외부 표현을 맡는다. 쿼리셋은 게으르지만 예측 가능해야 하고, 믹스인은 작되 신뢰할 수 있어야 하며, 신호는 드물게·명확하게만 울려야 한다. 트랜잭션과 멱등성은 안전을, 프로파일링과 캐시는 성능을, 타입 힌트와 테스트는 유지보수를 담보한다. 이 원칙을 작은 기능 하나에라도 적용해 나가면, 코드베이스는 읽기 쉬워지고 테스트 가능해지며, 성능과 신뢰성은 자연스럽게 따라온다. 결국 객체지향은 장식이 아니라 실전에 필요한 생존 전략이고, 장고는 그 전략을 구현하기 쉬운 발판을 제공한다. 이 글이 그 발판 위에서 한 걸음 더 멀리, 더 안정적으로 나아가는 데 실질적인 길잡이가 되길 바란다.