**Общая структура проекта(приложений)**

project/

settings/ # base/dev/prod конфиги

apps/

users/ # auth, профили, токены

billing/ # тарифы, webhooks

content/ # статьи/страницы/новости

core/ # общие утилиты/декораторы

<domain app>/ # предметные области (orders, catalog, etc.)

#-----Пример конкретного приложения-----

models.py # доменные сущности

services.py # прикладные/доменные сервисы

selectors.py # «чтение» (CQRS-подход, опционально)

tasks.py # Celery-задачи

api/

serializers.py

views.py (или viewsets.py)

urls.py

permissions.py / policies.py

tests/ # unit + api

#-----Пример конкретного приложения-----

urls.py

asgi.py / wsgi.py

— ключевая мысль: **ViewSet должен быть тонким**, бизнес-правила живут в services.py/«доменных сервисах», а сложные SELECT’ы — в selectors.py (если придерживаться лёгкого CQRS: «отдельно читаем, отдельно пишем»).

**Общая идея слоёв (конкрет. приложение)**

1. **Model** - гарантирует *инварианты базы* (unique email, not null).
2. **Serializer** - проверяет, что входные данные *валидны по форме* (email, пароль ≥ 6 символов, формат).
   * Вход: валидирует и нормализует JSON → Python-данные.
   * Выход: превращает доменные объекты/DTO → JSON.
   * Здесь — *формальная проверка формы и простых правил* (длина, формат, choices, object-level validate).
3. **View** - **ТОНКИЙ** отвечает за то, *как* это доступно через API.
4. **Service** - **ТОЛСТЫЙ**. Это место, где живёт **бизнес-логика**: правила, последовательности действий, транзакции, идемпотентность, вызовы внешних систем.
   * Транзакции (@atomic), инварианты, идемпотентность, побочные эффекты (WS/email/Celery — через on\_commit).
   * Возвращает доменные объекты или **DTO/dataclass**.
5. **Selector - “запросы/чтение”: вытаскивают данные оптимальными SQL**
   * Чистые запросы к БД ( .only, select\_related, фильтры/пагинация/сортировки ).
   * Без побочных эффектов. Возвращает QuerySet или **DTO** (dict/list).

**Сервис**

**Общие правила**

1. Всегда думай о **границах транзакций** (@transaction.atomic).
2. **Идемпотентность** там, где риск дублей (уникальные индексы + обработка IntegrityError).
3. Сложные инварианты — на уровне модели **и** в сервисе.
4. Внешние эффекты (WS, email, celery.delay) — оборачивай и выноси в on\_commit, чтобы не слали события до фикса транзакции.

**Когда сервис реально нужен**

* **Мульти-модельные операции** (user + profile + event + celery).
* **Идемпотентность** (POST /attempts, POST /payments/charge).
* **Сложные бизнес-правила** (например, streak: не инкрементировать 2 раза за день).
* **Побочные эффекты** (ws-push, email) → только через on\_commit.

**@dataclass**

@dataclass - Это декоратор из Python (from dataclasses import dataclass).

Он превращает обычный класс в «контейнер данных» (как DTO), автоматически добавляя:

* \_\_init\_\_ (конструктор),
* \_\_repr\_\_,
* сравнение и т.д.

Пример:

@dataclass

class RegisterResult:

user: User

is\_new: bool

Вместо того, чтобы возвращать «сложный кортеж» типа (user, True) или «сырой dict», мы получаем **читаемую структуру**.

Теперь:

res = register\_user(email="x@y.z", password="123")

print(res.user, res.is\_new)

Зачем это в сервисах:

* возвращать **понятный результат** (сразу видно поля, IDE подсказывает типы),
* легче тестировать (можно сравнивать объекты, а не словари),
* код становится самодокументируемым.

**Транзакции**

**Варианты реализации транзакций на примере register\_user**

**Вариант A: внутренний atomic только вокруг «рискованного» create**

(после отката savepoint можно продолжать и сделать get)

from dataclasses import dataclass

from django.db import transaction, IntegrityError

from django.contrib.auth import get\_user\_model

User = get\_user\_model()

@dataclass

class RegisterResult:

user: User

is\_new: bool

def register\_user(\*, email: str, password: str, locale: str) -> RegisterResult:

email = email.strip().lower()

# внешний код БЕЗ atomic (или может быть общий atomic выше по стеку)

try:

# локальный savepoint только на create\_user

with transaction.atomic():

user = User.objects.create\_user(

username=email.split("@")[0],

email=email,

password=password,

locale=locale,

)

return RegisterResult(user=user, is\_new=True)

except IntegrityError:

# мы ВНЕ того savepoint → уже можно делать запросы

user = User.objects.get(email=email)

return RegisterResult(user=user, is\_new=False)

**Вариант B: оставить внешний @transaction.atomic, но добавить вложенный atomic (savepoint)**

from django.db import transaction, IntegrityError

@transaction.atomic

def register\_user(\*, email: str, password: str, locale: str) -> RegisterResult:

email = email.strip().lower()

try:

with transaction.atomic(): # ВНУТРЕННИЙ savepoint

user = User.objects.create\_user(

username=email.split("@")[0], email=email, password=password, locale=locale

)

return RegisterResult(user=user, is\_new=True)

except IntegrityError:

# Вышли из внутреннего блока (откат к savepoint прошёл) → дальше можно читать

user = User.objects.get(email=email)

return RegisterResult(user=user, is\_new=False)

**Вариант C: get\_or\_create (самый простой для email-уникальности)**

def register\_user(\*, email: str, password: str, locale: str) -> RegisterResult:

email = email.strip().lower()

user, created = User.objects.get\_or\_create(

email=email,

defaults={"username": email.split("@")[0], "locale": locale}

)

if created:

user.set\_password(password)

user.save(update\_fields=["password"])

return RegisterResult(user=user, is\_new=created)

У get\_or\_create есть встроенная защита от гонок на уровне транзакции; на практике это самый простой и безопасный путь для «создать или вернуть существующего».

**Идемпотентность**

**Идемпотентность** в API/сервисах = свойство операции, при котором **многократный одинаковый вызов даёт тот же результат, что и один**.

* Пример **без идемпотентности**:

POST /api/attempts {score: 10} дважды подряд начислит XP два раза → дубли.

* Пример **с идемпотентностью**:

тот же POST с ключом idempotency\_key="abc123" → первый вызов создаст запись и начислит XP, второй вернёт тот же результат, без повторного начисления.

Это нужно:

* при **сетевых сбоях** (клиент может повторно послать запрос);
* при **повторных нажатиях** в UI;
* при **ретраях** со стороны фронта/очереди.

В Django обычно делают так:

* модель Attempt имеет unique\_together(user\_id, idempotency\_key);
* сервис ловит IntegrityError, возвращает старый результат вместо повторного начисления.

**Селекторы**

**Общие правила:**

1. Только чтение. Никаких save()/update()/внешних вызовов.
2. Агрессивно оптимизируй: .only(), .select\_related(), .prefetch\_related().
3. Возвращай либо QuerySet (для дальнейших слоёв), либо **плоские DTO** (dict/list) для API.
4. Удобно держать “репорты/агрегаты” и “списки с фильтрами” именно тут.

**Оптимизация запросов**

select\_related, prefetch\_related

* .only() - селектить только отдельные поля

Мини-пример, где only() стреляет в ногу

# Берём только id и email

user = User.objects.only("id", "email").get(id=uid)

# Сериализатор (или \_\_str\_\_) обращается к username → ORM делает ДОзагрузку

data = {"email": user.email, "name": user.username} # +1 запрос

.defer() - исключаешь определенные поля

only(), defer() Использовать когда получение кучи записей, тяжелые поля, много строк

**Практические рекомендации по «правильному» Django/DRF**

1. **Тонкие viewsets, толстые сервисы**
   * services.py: команды (create\_order, register\_user, grant\_xp).
   * selectors.py: чтение (get\_user\_with\_stats, list\_posts).
   * Вьюхи только валидируют вход и зовут эти функции.
2. **Границы транзакций**
   * Сервис отвечает за atomic() и последовательность.
   * Важные инварианты — на уровне модели (unique, constraints, clean()).
3. **Идемпотентность и гонки**
   * Уникальные индексы + get\_or\_create/конфликт-обработка.
   * Для критичных операций — idempotency-key.
4. **Конфиги/секреты**
   * 12-factor: .env, разные settings, не хардкодить секреты.
5. **OpenAPI как контракт**
   * drf-spectacular → схема → (опц.) автоген SDK/контракт-тесты.
6. **Тесты по пирамиде**
   * Unit для сервисов/правил; API-контракты для публичных эндпоинтов; немного smoke.
7. **Фоновые задачи**
   * Celery для долгих/периодических; **идемпотентность задач** (можно выполнить дважды).
8. **Реал-тайм**
   * Channels для уведомлений/чата; при росте — отдельный real-time сервис.

**Примеры для Users, Blog/News, Attempts**

**1) Users: регистрация и профиль**

[**serializers.py**](http://serializers.py)

# apps/users/api/serializers.py

from dataclasses import dataclass

from django.contrib.auth import get\_user\_model

from rest\_framework import serializers

User = get\_user\_model()

class RegisterIn(serializers.Serializer):

email = serializers.EmailField()

password = serializers.CharField(min\_length=6, write\_only=True)

locale = serializers.CharField(max\_length=8, required=False, default="en")

def validate\_email(self, v):

return v.strip().lower()

@dataclass

class RegisterDTO:

id: int

email: str

is\_new: bool

class RegisterOut(serializers.Serializer):

id = serializers.IntegerField()

email = serializers.EmailField()

is\_new = serializers.BooleanField()

class MeOut(serializers.ModelSerializer):

class Meta:

model = User

fields = ("id", "email", "username", "locale", "tz")

class MePatchIn(serializers.Serializer):

username = serializers.RegexField(r"^[a-z0-9\_]{3,20}$", required=False)

locale = serializers.CharField(max\_length=8, required=False)

tz = serializers.CharField(max\_length=64, required=False)

[**services.py**](http://services.py)

# apps/users/services.py

from dataclasses import dataclass

from django.db import transaction, IntegrityError

from django.contrib.auth import get\_user\_model

User = get\_user\_model()

@dataclass

class RegisterResult:

user: User

is\_new: bool

@transaction.atomic

def register\_user(\*, email: str, password: str, locale: str) -> RegisterResult:

try:

user = User.objects.create\_user(

username=email.split("@")[0], email=email, password=password, locale=locale

)

return RegisterResult(user=user, is\_new=True)

except IntegrityError:

user = User.objects.get(email=email)

return RegisterResult(user=user, is\_new=False)

@transaction.atomic

def update\_profile(\*, user: User, username=None, locale=None, tz=None) -> User:

if username is not None:

user.username = username

if locale is not None:

user.locale = locale

if tz is not None:

user.tz = tz

user.full\_clean()

user.save(update\_fields=["username", "locale", "tz"])

return user

[**selectors.py**](http://selectors.py)

# apps/users/selectors.py

from django.contrib.auth import get\_user\_model

User = get\_user\_model()

def get\_user\_by\_id(user\_id: int) -> User:

return User.objects.only("id", "email", "username", "locale", "tz").get(id=user\_id)

[**views.py**](http://views.py)

# apps/users/api/views.py

from rest\_framework import views, permissions, status

from rest\_framework.response import Response

from ..services import register\_user, update\_profile

from ..selectors import get\_user\_by\_id

from .serializers import RegisterIn, RegisterOut, RegisterDTO, MeOut, MePatchIn

class RegisterView(views.APIView):

permission\_classes = [permissions.AllowAny]

def post(self, request):

s\_in = RegisterIn(data=request.data); s\_in.is\_valid(raise\_exception=True)

res = register\_user(\*\*s\_in.validated\_data)

dto = RegisterDTO(id=res.user.id, email=res.user.email, is\_new=res.is\_new)

return Response(RegisterOut(dto).data,

status=status.HTTP\_201\_CREATED if res.is\_new else status.HTTP\_200\_OK)

class MeView(views.APIView):

permission\_classes = [permissions.IsAuthenticated]

def get(self, request):

u = get\_user\_by\_id(request.user.id)

return Response(MeOut(u).data)

def patch(self, request):

s\_in = MePatchIn(data=request.data); s\_in.is\_valid(raise\_exception=True)

u = update\_profile(user=request.user, \*\*s\_in.validated\_data)

return Response(MeOut(u).data)

Здесь InputSerializer нормализует email и проверяет пароль, Service делает транзакцию/идемпотентность, OutputSerializer превращает DTO/модель в JSON.

**2) Blog/News: фильтры/поиск → селектор; админ-CRUD → сервис**

[**serializers.py**](http://serializers.py)

# apps/blog/api/serializers.py

from rest\_framework import serializers

from ..models import Post

class PostListQuery(serializers.Serializer):

q = serializers.CharField(required=False)

tag = serializers.CharField(required=False)

ordering = serializers.ChoiceField(choices=["-published\_at", "published\_at"], default="-published\_at")

class PostOut(serializers.ModelSerializer):

class Meta:

model = Post

fields = ("id","title","slug","published\_at","updated\_at","tags")

class PostDetailOut(serializers.ModelSerializer):

class Meta:

model = Post

fields = ("id","title","slug","body\_md","published\_at","updated\_at","tags")

class PostCreateIn(serializers.Serializer):

title = serializers.CharField(max\_length=200)

body\_md = serializers.CharField()

tags = serializers.ListField(child=serializers.CharField(), required=False, default=list)

publish = serializers.BooleanField(default=False)

[**selectors.py**](http://selectors.py)

# apps/blog/selectors.py

from django.db.models import Q, QuerySet

from .models import Post

def list\_posts(\*, q=None, tag=None, ordering="-published\_at") -> QuerySet:

qs = Post.objects.filter(is\_published=True).only(

"id","title","slug","published\_at","updated\_at","tags"

)

if q:

qs = qs.filter(Q(title\_\_icontains=q) | Q(body\_md\_\_icontains=q))

if tag:

qs = qs.filter(tags\_\_contains=[tag])

return qs.order\_by(ordering)

def get\_post\_published(slug: str) -> Post:

return Post.objects.get(slug=slug, is\_published=True)

[**services.py**](http://services.py)

# apps/blog/services.py

from django.db import transaction

from django.utils import timezone

from django.utils.text import slugify

from .models import Post

@transaction.atomic

def create\_post(\*, title: str, body\_md: str, tags: list[str], publish: bool) -> Post:

post = Post.objects.create(

title=title, slug=slugify(title), body\_md=body\_md,

tags=tags or [], is\_published=publish,

published\_at=timezone.now() if publish else None

)

return post

[**views.py**](http://views.py)

# apps/blog/api/views.py

from rest\_framework import generics, permissions

from .serializers import PostListQuery, PostOut, PostDetailOut, PostCreateIn

from ..selectors import list\_posts, get\_post\_published

from ..services import create\_post

class PostListView(generics.ListAPIView):

permission\_classes = [permissions.AllowAny]

serializer\_class = PostOut

def get\_queryset(self):

qs\_in = PostListQuery(data=self.request.query\_params); qs\_in.is\_valid(raise\_exception=True)

return list\_posts(\*\*qs\_in.validated\_data)

class PostDetailView(generics.RetrieveAPIView):

permission\_classes = [permissions.AllowAny]

serializer\_class = PostDetailOut

lookup\_field = "slug"

def get\_object(self):

return get\_post\_published(self.kwargs["slug"])

class PostCreateView(generics.CreateAPIView):

permission\_classes = [permissions.IsAdminUser]

serializer\_class = PostDetailOut # для ответа

def create(self, request, \*args, \*\*kwargs):

s\_in = PostCreateIn(data=request.data); s\_in.is\_valid(raise\_exception=True)

post = create\_post(\*\*s\_in.validated\_data)

return self.get\_response(post)

def get\_response(self, post):

return self.response\_class(self.serializer\_class(post).data, status=201)

На read-эндпоинтах view напрямую «кормит» QuerySet в OutputSerializer. На write — InputSerializer → service → OutputSerializer.

**3) Attempts: входная валидация сериализатором, идемпотентность в сервисе, агрегации из селектора, пуш через on\_commit**

[**serializers.py**](http://serializers.py)

# apps/game\_core/api/serializers.py

from dataclasses import dataclass

from rest\_framework import serializers

class AttemptIn(serializers.Serializer):

idempotency\_key = serializers.RegexField(r"^[a-zA-Z0-9.\_-]{6,64}$")

game\_slug = serializers.CharField(max\_length=32)

kind = serializers.ChoiceField(choices=["round","task","session","harvest","quiz"])

success = serializers.NullBooleanField(required=False)

score = serializers.IntegerField(min\_value=0, required=False)

time\_spent\_ms = serializers.IntegerField(min\_value=0, max\_value=8\*60\*60\*1000, required=False)

@dataclass

class AttemptOutDTO:

xp\_delta: int

total\_xp: int

streak: int

created\_new: bool

class AttemptOut(serializers.Serializer):

xp\_delta = serializers.IntegerField()

total\_xp = serializers.IntegerField()

streak = serializers.IntegerField()

[**services.py**](http://services.py)

# apps/game\_core/services.py

from dataclasses import dataclass

from django.db import transaction, IntegrityError

from django.db import transaction as dj\_trx

from .models import Attempt, UserProfile

def \_xp(success, score) -> int:

if not success: return 0

s = score or 0

return min(10, 1 + s // 10)

def \_update\_streak(p: UserProfile):

# упрощённо: инкремент при первом сегодня

from django.utils.timezone import now

if not p.last\_active\_at or p.last\_active\_at.date() != now().date():

p.streak += 1

p.last\_active\_at = now()

@dataclass

class AttemptResult:

xp\_delta: int

total\_xp: int

streak: int

created\_new: bool

def send\_progress\_update(user\_id: int, xp\_delta: int, total\_xp: int, streak: int):

from asgiref.sync import async\_to\_sync

from channels.layers import get\_channel\_layer

async\_to\_sync(get\_channel\_layer().group\_send)(

f"user\_{user\_id}",

{"type": "progress.updated", "xp\_delta": xp\_delta, "total\_xp": total\_xp, "streak": streak},

)

@transaction.atomic

def record\_attempt(\*, user\_id: int, idempotency\_key: str, game\_slug: str, kind: str,

success=None, score=None, time\_spent\_ms=None) -> AttemptResult:

try:

attempt = Attempt.objects.create(

user\_id=user\_id, idempotency\_key=idempotency\_key,

game\_slug=game\_slug, kind=kind, success=success, score=score, time\_spent\_ms=time\_spent\_ms

)

created\_new = True

except IntegrityError:

attempt = Attempt.objects.get(user\_id=user\_id, idempotency\_key=idempotency\_key)

created\_new = False

profile = UserProfile.objects.select\_for\_update().get(user\_id=user\_id)

xp\_delta = \_xp(attempt.success, attempt.score)

if created\_new:

profile.xp += xp\_delta

\_update\_streak(profile)

profile.save(update\_fields=["xp","streak","last\_active\_at"])

res = AttemptResult(xp\_delta=xp\_delta, total\_xp=profile.xp, streak=profile.streak, created\_new=created\_new)

dj\_trx.on\_commit(lambda: send\_progress\_update(user\_id, res.xp\_delta, res.total\_xp, res.streak))

return res

[**selectors.py**](http://selectors.py) **(пример чтения агрегатов)**

# apps/game\_core/selectors.py

from django.db.models import Count

from .models import Attempt, UserProfile

def get\_profile(user\_id: int) -> UserProfile:

return UserProfile.objects.only("user\_id","xp","streak","last\_active\_at").get(user\_id=user\_id)

def attempts\_by\_game(user\_id: int) -> dict[str,int]:

rows = (Attempt.objects.filter(user\_id=user\_id)

.values("game\_slug").annotate(c=Count("id")))

return {r["game\_slug"]: r["c"] for r in rows}

[**views.py**](http://views.py)

# apps/game\_core/api/views.py

from rest\_framework import views, permissions, status

from rest\_framework.response import Response

from .serializers import AttemptIn, AttemptOut, AttemptOutDTO

from ..services import record\_attempt

class AttemptView(views.APIView):

permission\_classes = [permissions.IsAuthenticated]

def post(self, request):

s\_in = AttemptIn(data=request.data); s\_in.is\_valid(raise\_exception=True)

res = record\_attempt(user\_id=request.user.id, \*\*s\_in.validated\_data)

dto = AttemptOutDTO(xp\_delta=res.xp\_delta, total\_xp=res.total\_xp, streak=res.streak, created\_new=res.created\_new)

return Response(AttemptOut(dto).data,

status=status.HTTP\_201\_CREATED if res.created\_new else status.HTTP\_200\_OK)

Тут входная бизнес-валидация (форматы, границы, choices) — в AttemptIn.

**Инварианты записи/гонки/идемпотентность** — в service.

**Формат ответа** — AttemptOut, который сериализует DTO.

**Разделение services и serializers**

**Есть два каноничных способа(НЕ СМЕШИВАТЬ):**

**Вариант A. «Тонкий сериалайзер» + явный вызов сервиса из вьюхи**

(как у меня в примере)

# serializers.py — только валидация формы

class PostCreateIn(serializers.Serializer):

title = serializers.CharField(max\_length=200)

body\_md = serializers.CharField()

tags = serializers.ListField(child=serializers.CharField(), required=False, default=list)

publish = serializers.BooleanField(default=False)

# views.py — валидируем вход и зовём сервис

def create(self, request, \*args, \*\*kwargs):

s\_in = PostCreateIn(data=request.data)

s\_in.is\_valid(raise\_exception=True)

post = create\_post(\*\*s\_in.validated\_data) # <-- services.py

return Response(PostDetailOut(post).data, status=201)

**Плюсы:** сериалайзеры не жирнеют, бизнес-операции централизованы; в сервисах легко ставить atomic, on\_commit, идемпотентность.

**Минусы:** вьюхи чуть многословнее; нельзя «из коробки» полагаться на serializer.save().

**Вариант B. Делегировать из create() сериалайзера в сервис**

(компромисс: DRF API привычно, логика — в сервисах)

# serializers.py

class PostCreateSerializer(serializers.Serializer):

title = serializers.CharField(...)

body\_md = serializers.CharField(...)

tags = serializers.ListField(...)

publish = serializers.BooleanField(default=False)

def create(self, validated\_data):

from apps.blog.services import create\_post

return create\_post(\*\*validated\_data) # <-- вся логика всё равно в сервисе

# views.py — можно использовать CreateAPIView

class PostCreateView(generics.CreateAPIView):

serializer\_class = PostCreateSerializer

**Плюсы:** совместимо с generic-вьюхами; бизнес-логика всё ещё в сервисе.

**Минусы:** сериалайзер знает о сервисах (не идеально чистая зависимость, но практично).

Я обычно выбираю Вариант B для CRUD, и Вариант A — для сложных юзкейсов (несколько моделей, побочные эффекты, идемпотентность).

**Где делать валидацию?**

* **В сериалайзере**: форматы, типы, длины, choices, нормализация (email.lower()), кросс-полевые проверки в validate(). Это «валидация на границе» (API-контракт).
* **В сервисе**: доменные инварианты и последовательности (транзакции, идемпотентность, создание нескольких сущностей, on\_commit для email/WS/Celery).
* **В модели/БД**: то, что **обязательно** всегда (уникальные индексы, CHECK, NOT NULL).

Таким образом, даже если «ничего не создаётся в сериалайзере», **валидация всё равно делается в нём** — он чистит и проверяет вход перед вызовом сервиса:

s\_in = RegisterIn(data=request.data) # сериалайзер валидирует и нормализует

s\_in.is\_valid(raise\_exception=True)

res = register\_user(\*\*s\_in.validated\_data) # сервис делает транзакцию и побочк

**Три мини-примера**

**1) Простая запись (можно прямо в сериалайзере)**

class TagSerializer(serializers.ModelSerializer):

class Meta:

model = Tag

fields = ("name",)

def create(self, validated\_data):

return Tag.objects.create(\*\*validated\_data)

**2) Делегирование в сервис (CRUD-совместимо)**

class RegisterSerializer(serializers.Serializer):

email = serializers.EmailField()

password = serializers.CharField(min\_length=6, write\_only=True)

def validate\_email(self, v): return v.strip().lower()

def create(self, validated\_data):

from apps.users.services import register\_user

res = register\_user(\*\*validated\_data) # atomic, profile, event, on\_commit email

return res.user

Теперь CreateAPIView может вызывать serializer.save(), а бизнес-логика остаётся в сервисе.

**3) Сложный юзкейс (лучше не через save())**

class AttemptIn(serializers.Serializer):

idempotency\_key = serializers.RegexField(r"^[\\w.-]{6,64}$")

game\_slug = serializers.CharField(max\_length=32)

kind = serializers.ChoiceField(choices=["harvest","quiz","round"])

success = serializers.NullBooleanField(required=False)

score = serializers.IntegerField(min\_value=0, required=False)

class AttemptView(APIView):

def post(self, request):

s = AttemptIn(data=request.data); s.is\_valid(raise\_exception=True)

res = record\_attempt(user\_id=request.user.id, \*\*s.validated\_data) # services.py: atomic, idempotent, on\_commit WS

return Response({"xp\_delta": res.xp\_delta, "total\_xp": res.total\_xp}, status=201 if res.created\_new else 200)

Здесь **специально** не используем serializer.save(), чтобы явно показать orchestration и статус 200/201 по created\_new.