**Подробные описания всех приложений**

**1. ☑️ Game\_core — ядро (пользователи, XP, прогресс)**

Модели:

1. ✅Attempt\*(id, user\_id, game\_slug:str, kind:str, success:bool|null, score:int|null, time\_spent\_ms: int|null, created\_at: DateField)\*
   * Attempt fields:
     + id, user\_id
     + idempotency\_key: *charfield 64symbols*

Это просто **уникальный идентификатор операции**, который генерируется на клиенте (игрой) и отправляется вместе с запросом: *"idempotency\_key": "42e7f94c-19c5-4b07-b3a6-0e6c54f621af"*

* + - unique\_together = (user\_id, idempotency\_key)
    - game\_slug *# какая мини-игра*
    - kind *# 'round' | 'task' | 'session' | 'challenge'* ***CHOICES****!!*

kind — чтобы отличать типы активности: «сессия», «раунд», «задача». В одном проекте мини-игры могут быть очень разные, так проще фильтровать.

Нормализовать словарь **kind.**

Сейчас свободная строка. Мини-предотвращение хаоса:

заведи Enum/choices на бэке (например: round|task|session|quiz|harvest|kill|craft|trade),

на фронте держи такой же список в одном месте. Если появится экзотика — расширишь.

* + - success: bool|null *# null если неприменимо (песочница)*
    - score: int|null *# результат игры(очки/ответы/время)*

XP считать на бэке по правилу из score и/или success , и возвращать xp\_delta в ответе(websocket). Так код станет прозрачнее.

* + - time\_spent\_ms: int|null

Время, затраченное на то или иное действие. Для MVP:

* + - * В ферме: harvest → ставить time\_spent\_ms = null или 2000 (если хочется «правдоподобности»).
      * В боевой мини-игре: kill → фиксировать реальную длительность боя, если она есть.
      * Для всех остальных — поле оставить nullable и заполнять только когда это имеет смысл.

На фронте при начале действия запоминаешь started\_at (в памяти).

На завершении делаешь **один** POST /api/attempts с:

* + - * game\_slug, kind, success, score, meta
      * time\_spent\_ms = (ended\_at - started\_at) (рассчитан на клиенте)
    - meta: JSONB *# что угодно от игры (например {words\_correct: 7})*

Оно позволяет каждой игре прикладывать свои данные, не меняя схему базы.

Например:

{words\_correct:7, words\_total:10};

{path:[{x:1,y:2},{x:3,y:4}]};

{difficulty:'hard'};

{ "crop": "carrot", "amount": 3, "field\_id": "tile\_12" };

{ "mob": "wolf", "weapon": "sword", "location": "forest\_edge" };

Если этого не будет, придётся постоянно мигрировать базу при добавлении новых типов игр

* + - created\_at

Индексы:

(user\_id, created\_at DESC) — профиль/ленты событий;

(user\_id, game\_slug, created\_at DESC) — фильтр по игре;

* + Валидация

Проверяй success ∈ {true,false,null},

score >= 0 (если пришёл),

time\_spent\_ms в разумных пределах (например, 0..8h).

* + Обработка

Приходит запрос с idempotency\_key.

Бэк проверяет в БД:

есть ли уже запись для (user\_id, idempotency\_key).

Если есть → возвращает **тот же результат**, что в первый раз.

Если нет → создаёт новую Attempt, сохраняет ключ, считает XP.

if Attempt.objects.filter(user=user, idempotency\_key=key).exists():

return previous\_response

else:

create Attempt(...)

return new\_response

* + Json Examples
    - Ферма

{ "user\_id": 42, "game\_slug": "farm", "kind": "harvest", "success": true, "score": 1, "time\_spent\_ms": 3000, "meta": { "crop": "carrot", "amount": 3, "field\_id": "tile\_12" } }

* + - Убийство моба

{ "user\_id": 42, "game\_slug": "farm", // или "rpg" если это другая мини-игра "kind": "kill", "success": true, "score": 10, "time\_spent\_ms": 8000, "meta": { "mob": "wolf", "weapon": "sword", "location": "forest\_edge" } }

* + - Рефлекс-кликер (реакция/тайминг)

{ "user\_id": 42, "game\_slug": "reaction", "kind": "round", "success": true, "score": 8, "time\_spent\_ms": 52000, "meta": { "hits": 8, "misses": 2, "avg\_latency\_ms": 240 } }

* + - Песочница/строительство (нет «победы»)

{ "user\_id": 42, "game\_slug": "builder", "kind": "session", "success": null, "score": null, "time\_spent\_ms": 180000, "meta": { "placed\_blocks": 87, "blueprint\_id": "barn\_v1" } }

* + Действия после:

После POST /api/attempts сервер отправляет WS-пуш progress.updated

{ "type": "progress.updated", "xp\_delta": 10, "total\_xp": 330, "streak": 7 }

Стрик: инкремент при первом Attempt в сутки.

События WS: progress.updated (поднимет Channels, см. ниже).

**Расширить позже:** формулы XP, бейджи, снапшоты лидербордов, античит.

if success == true:

xp\_delta = min(10, 1 + floor(score / 10)) # мягкая шкала

else:

xp\_delta = 0

1. ✅UserProfile\*(user OneToOne, xp:int=0, streak:int=0, level:int=1, updated\_at, ...)\*

МБ СРАЗУ РАЗДЕЛИТЬ НА ПРОФИЛЬ/НАСТРОЙКИ/ПРОГРЕСС ИГР

* + UserProfile fields:
  + UserProfile
  + - user FK -> auth.User (OneToOne) [PK]
  + - xp int # общий опыт
  + - level int # если нужен сейчас; иначе всегда 1
  + - streak int # текущая серия активных дней
  + - last\_active\_at datetime # для пересчёта streak
  + - ui\_lang char(8) # "en", "ru", ...
  + - notifications bool # вкл/выкл
  + - plan\_key str|null # "free"|"pro" и т.п. (синхронизируется с payments)
  + - created\_at datetime
  + - updated\_at datetime
  + # агрегаты
  + - games\_played int # (агрегат из /attempts)кол-во уникальных game\_slug с попытками
  + - attempts\_total int # (агрегат из /attempts) всего попыток (успешных + нет)
  + - words\_learned int # из лексики (уходит в профиль)
  + # Сколько лексем добавлено в словарь.
  + # Формально это count(distinct lexeme\_id) в user\_vocab; для скорости можно кэшировать числом и периодически сверять.
  + Services

обновляются только сервисами

* + - **ProfileProgressService** — обновляет xp/streak/attempts\_total/games\_played. В UserProfile добавь (или упомяни) tz (например, Europe/Helsinki). Streak считай по этой TZ, но все даты храни в UTC
    - **ProfileSettingsService** — меняет ui\_lang/notifications.
    - **BillingService** — читает/ставит plan\_key (до миграции на payments.subscription).
  + endpoints
    - GET /api/gameprofile/[id]
    - {
    - "xp": 320,
    - "level": 4,
    - "streak": 6,
    - "games\_played": 3,
    - "attempts\_total": 57,
    - "words\_learned": 24,
    - "ui\_lang": "ru",
    - "notifications": true,
    - "plan\_key": "free",
    - "last\_active\_at": "2025-09-19T13:20:12Z"
    - }
    - PATCH /api/gameprofile/[id]
    - { "ui\_lang": "en", "notifications": false }

1. ❌\*(опц.)\* Achievement\*(key, title)\*, UserAchievement\*(user, key, earned\_at)\*
   * Achivement fields
   * Achievement:
   * - key # 'first\_attempt', 'farm\_10\_rounds', ...
   * - scope: 'global' | 'game'
   * - game\_slug: null | 'farm' # для scope='game'
   * - title, description
   * К твоим — добавь enabled: bool и hidden: bool (потом пригодится для секреток и выключения без удаления).
2. ❌\*(опц.)\* GET /api/leaderboard?period=weekly|all

**2.✅ Users — аутентификация и профили**

**Модель:**

users.CustomUser (основана на AbstactUser)

* CustomUser fields
* id: UUIDField, pk
* email: EmailField(unique=True)
* username: CharField(unique=True, null/blank=True)
* display\_name: CharField(null=False, blank=True)
* first\_name, last\_name: CharField(null/blank=True)
* avatar\_url: URLField/imgfield(null/blank=True)
* locale: Char(8), default='ru'
* tz: Char(64), default='UTC'
* email\_verified: BooleanField(default=False)
* oauth\_provider: CharField(null/blank=True) / oauth\_sub: CharField(null/blank=True)
* marketing\_consent: BooleanField(default=False)
* privacy\_consent\_at: DateTimeField(null=True)
* deleted\_at: DateTimeField(null=True)
  + Объяснения:
    - id: UUIDField, pk

Удобно для внешних ссылок и приватности (не светим автоинкремент).

* + - email: EmailField(unique=True)

Единая точка входа. Делаем **уникальным** и нормализуем (lowercase).

* + - username: CharField(unique=True, nullable/blank=False) Оставляем, но **не требуем** в регистрации; можем автогенерить user\_8fj2…. Полезно для чата/никнейма.
    - first\_name, last\_name: CharField

Опционально. В форме регистрации можно не требовать.

* + - avatar\_url: URLField(null/blank=True)

Прямая ссылка на аватар (S3/статик). В MVP загружаем через presigned-URL из модуля media. Генерировать если не выбрана(буква от username)

* + - locale: Char(8), default='en'

Предпочтительный язык UI ('ru', 'en-US'). Дублирует UserProfile.ui\_lang, но держать тут удобно для письма/почты.

* + - tz: Char(64), default='UTC'

Таймзона для писем/напоминаний; streak считаем, скорее всего, по профилю — но у пользователя тоже не помешает.

* + - email\_verified: BooleanField(default=False)

Флажок подтверждения почты.

* + - oauth\_provider: CharField(null/blank=True) / oauth\_sub: CharField(null/blank=True)

Для входа через Google/GitHub (позже). Свяжем пару «провайдер + внешний id».

* + - marketing\_consent: BooleanField(default=False)

Согласие на рассылки.

* + - privacy\_consent\_at: DateTimeField(null=True)

Принятие политики/ToS (юридический след).

* + - deleted\_at: DateTimeField(null=True)

**Soft-delete** (GDPR-friendly): скрываем аккаунт, но можем выдержать период восстановления.

* Стандартные поля из AbstractUser, которые остаются:
  + password (хэшированный), is\_active, is\_staff, is\_superuser, last\_login, date\_joined.

**Индексы/ограничения**

* + Unique(email)
  + Unique(username) (если включён никнейм)
  + Индекс по oauth\_provider, oauth\_sub (комбинированный)
  + Индекс по deleted\_at (быстрее фильтровать активные)
* Эндпоинты (DRF)

**1) Регистрация POST /api/auth/register**

{ "email":"user@example.com", "password":"Secret123!", "locale":"ru" }

Ответ 201:

{ "id": "6b6d…", "email":"user@example.com", "email\_verified": false }

* + Создаём пользователя, отправляем письмо подтверждения (опц., можно отложить).
  + Если используешь сессии — сразу логиним, если требуется — ждём верификацию.

**2) Логин POST /api/auth/login**

* + Сессии: ставим httpOnly cookie.
  + JWT: отдаём access(+refresh).

**3) Логаут POST /api/auth/logout**

* + Сессии: session.flush().
  + JWT: заносим refresh в denylist (если включено).

**4) Профиль пользователя GET /api/profile/[id]**

{

"id":"6b6d…",

"email":"user@example.com",

"email\_verified": false,

"username":"user\_4f2a",

"avatar\_url": null,

"locale":"ru",

"tz":"Europe/Helsinki"

}

**5) Обновление профиля пользователя (личные поля) PATCH /api/profile**

{ "username":"totya", "avatar\_url":"https://...", "locale":"en", "tz":"Europe/Helsinki" }

* + Валидируем уникальность username, список допустимых locale/tz.

**6) Смена пароля POST /api/auth/change-password**

{ "current\_password":"...", "new\_password":"..." }

**7) Сброс пароля (двухходовый)**

* + POST /api/auth/password-reset — шлём письмо с токеном.
  + POST /api/auth/password-reset/confirm — { "token":"...", "new\_password":"..." }.

**8) Подтверждение почты (если включено)**

* + POST /api/auth/verify-email/request — выслать письмо.
  + POST /api/auth/verify-email/confirm — { "token":"..." } → email\_verified=true.

**9) Удаление аккаунта (soft-delete) POST /api/auth/delete**

* + Ставим deleted\_at=now(), is\_active=False.
  + (Опц.) cron удалит навсегда через N дней.

Дополнительно (позже): OAuth /api/auth/oauth/<provider> и refresh-ротация для JWT.

* Валидаторы:
  + email — lowercase/trim, формат валиден.
  + password — минимум 6-8 символов, стандартный Django validators
  + username — [a-z0-9\_]{3,20}; резервируем системные (admin, support, и т.д.).
  + locale — из белого списка (en, ru…). (CHOICES)
  + tz — IANA (например, Europe/Helsinki) из pytz/zoneinfo. (CHOICES мб)
  + deleted\_at is null для активных вызовов; не даём логиниться soft-deleted.
* Сериалайзеры:
  + RegisterSerializer(email, password, locale?)
  + LoginSerializer(email, password)
  + ProfileSerializer(id, email, email\_verified, username, avatar\_url, locale, tz)
  + ProfileUpdateSerializer(username?, avatar\_url?, locale?, tz?)
  + ChangePasswordSerializer(current\_password, new\_password)

После создания CustomUser после регистрации сигналом(post\_save) или вьюхой создавать UserProfile

**3. ❌ lexeme — лингвистическая библиотека (MVP: Lexeme + UserVocab)**

**Lexeme**

**Цель модуля**

* Давать подсказки к объектам мини-игр (тексты берём из **статических i18n-бандлов**).
* Позволять игроку добавлять слова в **аккаунтный словарь** (хранится на бэке).
* Никакой сложной морфологии/переводов на старте, только база для масштабирования.

**Lexeme, UserVocab Models**

# apps/lexeme/models.py

from django.conf import settings

from django.db import models

class Lexeme(models.Model):

id = models.CharField(primary\_key=True, max\_length=64) # "apple"

headword = models.CharField(max\_length=128) # "apple"

lang = models.CharField(max\_length=8, default="en") # базовый язык леммы

created\_at = models.DateTimeField(auto\_now\_add=True)

updated\_at = models.DateTimeField(auto\_now=True)

class UserVocab(models.Model):

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

lexeme = models.ForeignKey(Lexeme, on\_delete=models.CASCADE)

context = models.JSONField(default=dict, blank=True) # {game:"farm", object\_id:"tree\_1"}

created\_at = models.DateTimeField(auto\_now\_add=True)

class Meta:

unique\_together = ("user", "lexeme")

indexes = [

models.Index(fields=["user", "created\_at"]),

models.Index(fields=["lexeme"]),

]

* Lexeme.id — короткий стабильный идентификатор (совпадает с headword на MVP, позже можно мигрировать на slug/UUID).
* lang оставили на будущее (если захочешь учить не только EN).
* UserVocab.context — лёгкая «память» места, где игрок взял слово (для экранов повторения).

**Endpoints**

* POST api/vocab

Служит для добавления слова(+контекста откуда это слово) в словарь

*Request:*

{

"lexeme": "apple",

"context": {"game":"farm","object\_id":"tree\_1"}

}

*Response 200/201*

{

"lexeme": "apple",

"headword": "apple",

"added\_at": "2025-09-20T12:00:01Z"

}

Валидация:

* + lexeme обязателен, 1..64 символов, [a-z0-9.\_-] (рекомендуемый набор).
  + Если лексемы нет в таблице — опция:
    - DEV: автосоздать Lexeme(id=headword);
    - PROD: 400 + сообщение «unknown lexeme» (выберешь стратегию флагом ALLOW\_CREATE\_MISSING\_LEXEME).
* GET api/vocab?max\_on\_page=50&page=2

Служит для получения словаря пользователя

*Response*

{

"count": 2,

"results": [

{"lexeme":"apple","headword":"apple","added\_at":"2025-09-19T10:10:10Z"},

{"lexeme":"cow","headword":"cow","added\_at":"2025-09-19T10:12:33Z"}

]

}

* DELETE api/vocab[lexeme]

Удаляет слово из личного словаря

Response 204

* GET /api/vocab/{id}

получение конкретной записи в словаре

{"id":"apple","headword":"apple","lang":"en"}

**Фронт**

Подсказки /public/…/farm/en.json

{

"object.onion": "Onion",

"object.carrot": "Carrot",

"object.cow": "Cow",

"ui.add": "＋ Add"

}

Индексы объектов игры /games/farm/index.json

[

{ "id":"tile\_12", "i18nKey":"object.onion", "lexeme":"onion" },

{ "id":"tile\_13", "i18nKey":"object.carrot", "lexeme":"carrot" },

{ "id":"cow\_1", "i18nKey":"object.cow", "lexeme":"cow" },

{ "id":"ok", "i18nKey":"ui.ok" }

]

* i18nKey → текст подсказки (из бандла).
* lexeme → что добавлять в словарь при клике на «＋» (может отсутствовать для сугубо UI-строк).

**Хуки**

**useBundle(game, locale)**

* грузит и кэширует бандл,
* возвращает t(key) и isReady.

**useVocab()**

* list — GET /api/vocab,
* add(lexeme, context) — POST /api/vocab,
* remove(lexeme) — DELETE /api/vocab/{lexeme},
* optimistic-UI и тосты.

Псевдокод:

function onHover(obj) {

const label = t(obj.i18nKey) || obj.i18nKey;

showTooltip({

text: label,

actions: obj.lexeme

? [{ label: t("ui.add") || "＋", onClick: () => add(obj.lexeme, {game, object\_id: obj.id}) }]

: []

});

}

* UX
  + Десктоп: шорткат A при наведении равен клику на «＋».
  + Мобайл: long-press → меню «Добавить».
  + После успешного добавления: тост «Added to your vocab» и смена иконки на «✓».
* Поток
  + Игра рендерит объект cow\_1

→ находит в index.json: {i18nKey:"object.cow", lexeme:"cow"}.

* + При наведении:

t("object.cow") из бандла → показывает «Cow» + кнопку «＋».

* + Игрок кликает «＋»:

фронт вызывает:

POST /api/vocab

{

"lexeme": "cow",

"context": {"game":"farm","object\_id":"cow\_1"}

}

* + Бэк:
    - проверяет наличие Lexeme("cow") (создаёт в DEV при флаге),
    - создаёт UserVocab (или 409, если уже есть),
    - отдаёт 201/200 с данными.
  + Фронт:
    - помечает объект как «добавлено» (например, заменяет «＋» на «✓»),
    - useVocab().list обновляется (optimistic или через refetch).

**Примеры**

**5.1 Ферма — грядка с луком**

**Подсказка**: object.onion → «Onion»

**Добавление**:

POST /api/vocab

{ "lexeme":"onion", "context":{"game":"farm","object\_id":"tile\_12"} }

**5.2 Коровник — корова**

**Подсказка**: object.cow → «Cow»

**Добавление**:

POST /api/vocab

{ "lexeme":"cow", "context":{"game":"farm","object\_id":"cow\_1"} }

**5.3 UI-кнопка без лексемы**

**Подпись**: ui.ok → «OK»

**Добавление**: кнопки «＋» нет (нет lexeme).

**Доп. (бизнес,защита,админка)**

**Бизнес-правила и простая защита**

* **Троттлинг**: 60 req/min/user на /vocab.
* **Размер context**: до ~2–4 КБ.
* **Аудит**: сохраняем только факт и время; никакой PII из фронта в context.

**Админ и сидинг**

* Django admin для Lexeme, UserVocab.
* Команда manage.py import\_lexemes data/lexemes\_en.txt (по одной лемме в строке).
* Тестовый сид: apple, onion, carrot, cow, field, wolf, sword… (50–200 штук).

**Вкратце**

**один стол «лексемы» + хранение словаря игрока**

Когда брать: нужен аккаунтный словарь (не теряется), но без сложной контент-схемы.

**Идея**

* Подсказки всё ещё статические бандлы (как в B).
* Добавляем крошечный DRF-бэк с двумя сущностями: Lexeme(id, headword) и UserVocab(user\_id, lexeme\_id, context).
* Фронт по клику шлёт POST /vocab. Всё остальное как в B.

**Мини-модели**

# Lexeme: только id и базовая форма

class Lexeme(models.Model):

id = models.CharField(primary\_key=True, max\_length=64) # "apple"

headword = models.CharField(max\_length=128)

# UserVocab: что добавил игрок

class UserVocab(models.Model):

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

lexeme = models.ForeignKey(Lexeme, on\_delete=models.CASCADE)

context = models.JSONField(default=dict, blank=True) # {game:'farm', object\_id:'tree\_1'}

created\_at = models.DateTimeField(auto\_now\_add=True)

class Meta:

unique\_together = ('user','lexeme') # без дублей

**Эндпоинты**

* POST /vocab { lexeme:"apple", context:{game:"farm", object\_id:"tree\_1"} }
* GET /vocab — список слов игрока.

**Плюсы**: словарь синхронизируется между устройствами; простая база; тексты всё ещё дешёвые (статик-бандлы).

**Минусы**: нет сложной грамматики/переводов; но это можно добавить позже без ломки API.

**(LATER) Phrases and control by voice**

**1) Общий поток (MVP)**

1. Кліент: speech-to-text (на фронте) → текст "harvest the onions".
2. Бэкенд POST /nlp/parse:
   * нормализует текст,
   * извлекает **интент/действие** и **объект(ы)** + параметры,
   * возвращает структуру.
3. Кліент получает {action: "harvest", target\_lexeme: "onion", quantity: null, ...} и **сам** исполняет (находит подходящую цель рядом и выполняет).
   * (позже можно добавить POST /commands/execute, если часть правил должна проверяться/исполняться на сервере.)

**2) Контракт API (минимум)**

**Запрос**

POST /nlp/parse

{ "text": "harvest the onions", "locale": "en", "game": "farm", "player": {"x":12,"y":7} }

**Ответ (успех)**

{

"intent": "harvest",

"slots": {

"object": {"lexeme": "onion", "plural": true},

"quantity": null,

"location": null,

"filters": []

},

"confidence": 0.88,

"normalized": "harvest onion",

"notes": ["matched rule: VERB(harvest) + DET? + NOUN(onion\*)"]

}

**Ответ (нужно уточнение)**

{

"intent": "harvest",

"slots": {"object": {"candidates": ["onion","garlic"]}},

"confidence": 0.52,

"need\_confirmation": true,

"message": "Did you mean onions or garlic?"

}

**3) MVP-парсер (детерминированный, без ML)**

Построим на простых правилах — быстро и прозрачно.

**Шаги**

1. Нормализация: lowercasing, обрезка пунктуации, свёртка множественного числа к лемме (простая лемматизация: словарь → onions→onion).
2. Токенизация и стоп-слова (the, a, to, please).
3. **Verb→Intent**: таблица глаголов/синонимов для игры:
   * harvest|pick|gather → harvest
   * plant|sow → plant
   * craft|make → craft
   * equip|wear → equip
   * drop|discard → drop
   * attack|hit|kill → attack
4. **Noun→Lexeme**: словарь предметов/существ:
   * onion|onions|bulb onion → onion (lexeme\_id)
   * carrot|carrots → carrot
   * wolf|wolves → wolf
5. Шаблоны (регексы/правила) для базовых конструкций:
   * ^VERB NOUN(s)$ → {intent, object}
   * ^VERB (the|a)? NOUN(s) (from|in|at) NOUN/ID$ → {intent, object, location}
   * ^VERB NUMBER NOUN(s)$ → {intent, object, quantity}
   * отрицания: ^don't VERB → {intent, negated:true} (пока можно отвергать)
6. Построение слотов: {object, quantity, location, filters} и confidence:
   * базовая уверенность = вес правила + совпадение слов (0.6–0.9).
7. Если нет уверенного совпадения → вернуть need\_confirmation=true с подсказкой.

Это реально собирается за 1–2 файла кода: список глаголов/существ (для каждой игры), набор правил (несколько regex/pattern-матчеров), простая лемматизация, плюс маппинг на lexeme\_id из твоего модуля Lexeme.

**4) Разрешение объектов на фронте (исполнение)**

Клиент, получив {intent:"harvest", object:"onion"}, делает:

* **поиск цели** рядом: «есть ли грядка с lexeme=onion в радиусе R и в зоне видимости?»
* **валидации** (локальные или через лёгкий GET /world/query): не занято ли, нет ли кулдауна, принадлежит ли игроку.
* если несколько кандидатов — подобрать ближайший / спросить пользователя.
* выполнить действие: играть анимацию, после успеха отправить POST /attempts (как договорились).

**5) Список действий (стартовый словарь)**

Для фермы:

* harvest (собрать) — onion, carrot, wheat, …
* plant (посадить) — seed, onion, carrot
* water (полить) — field, crop
* fertilize (удобрить) — field/crop
* craft (скрафтить) — seed bag, hoe, scarecrow
* drop, pick up, use, open/close, store, sell, buy, trade

Для боёв/общей механики:

* attack (напасть/ударить/стрелять) — wolf, boar
* equip/unequip — sword, armor
* heal/drink — potion
* follow/talk/open (NPC/door)
* build/dismantle — barn, fence
* move (опц. для навигации): move north / go to barn (позже)

Список — это две маленькие таблицы на бэке: Intents (глаголы) и NounMap (синонимы → lexeme\_id) c привязкой к game\_slug.

**6) Валидация на бэке (когда нужно)**

Хотя исполнение локальное, бэк может делать «мягкую» проверку:

* **право/владение** (можно ли этому игроку собирать этот объект),
* **кулдауны/стамина**,
* **глобальные правила** (например, ночь — нельзя собирать цветы).

Механика:

* POST /nlp/parse → ответ с intent/slots.
* (опц.) POST /nlp/validate → {intent, slots, player\_state} → {ok:true|false, reason}.

**7) Расширяемость (дальше без ломки)**

* **Слоты**: добавляем tool, target\_player, direction, time и т.д.
* **Коллокации/фразы**: таблица GrammarPattern (шаблоны), напр. "harvest all onions on field {id}" (slot {id}).
* **Контекст**: «them/it/that» → ссылка на предыдущий object (диалоговая память на 1–2 шага).
* **Сложные цепочки**: "harvest onions and plant carrots" → два интента (пошаговое выполнение).
* **Мультиязычность**: отдельные словари глаголов/существ (ru→intent, en→intent) + сопоставление с lexeme\_id (это уже есть).
* **ML-режим** (когда появятся данные): лёгкий Intent-классификатор + слот-теги (CRF/transformer), но **поверх** тех же контрактов intent/slots. Можно гибрид: ML даёт гипотезу, правила проверяют и доводят.

**8) Хранилище данных (минимум)**

В модуле Lexeme/NLP:

* intents

id, slug (harvest), verbs\_by\_locale ({"en":["harvest","pick"],"ru":["собери","собрать"]}), game\_slug (null = глобальное).

* noun\_synonyms

form (onions), lexeme\_id (onion), locale, game\_slug?, priority.

* patterns (опц.)

locale, pattern (напр. ^(VERB) (NUMBER)? (NOUNS)$), slots (schema), intent.

* (можно и в JSON-конфиге для MVP; потом вынести в таблицы/админку)

**9) Обработка неоднозначностей/ошибок**

* Низкая уверенность → need\_confirmation=true, сообщение пользователю.
* Нет такого объекта рядом → {ok:false, reason:"no\_target\_in\_range"} (клиент покажет подсказку).
* Негативные/запретные команды → {ok:false, reason:"not\_allowed"}.
* Слишком частые команды → rate-limit (и для /attempts, и для /nlp/parse).

**10) Тест-наборы (что проверить в первых тестах)**

* harvest onions → {intent:"harvest", object:"onion"}
* pick the onion → то же
* harvest three onions → {quantity:3}
* harvest onions from field three → {location:"field\_3"}
* don't harvest → {negated:true} → отклонить
* kill the wolf with sword → {intent:"attack", object:"wolf", tool:"sword"}
* harvest onions and carrots → два интента или один с множественным объектом (решить политику заранее)

**Модели (ровно минимум, «Вариант C»):**

* Lexeme(id: str, headword: str, lang='en', created\_at)
* UserVocab(id, user FK, lexeme FK, context JSONB, created\_at)

*Уникальный индекс* (user, lexeme) — без дублей.

**Эндпоинты:**

* GET /api/lexemes/{id} → {id, headword, lang}
* GET /api/vocab → [{lexeme, headword, added\_at}]
* POST /api/vocab → {lexeme: "apple", context?: {...}}
* DELETE /api/vocab/{lexeme}

**4. ❌ game\_\* — приложения игр (валидация, свои модели)**

**Идея:** каждая мини-игра — отдельное Django-приложение при необходимости.

Каждое game\_\* — это «локальная логика конкретной мини-игры»: её правила, минимальные модели, валидация и (если надо) серверный стейт. Оно **не** занимается пользователями/XP/лексикой — только своей механикой — и общается с ядрами через понятные контракты:

* присылает результат в game\_core (POST /attempts);
* опционально читает/пишет свой небольшой стейт;
* переиспользует lexeme только на фронте (подсказки).

Дальше примеры для game\_farm

**Задачи game\_farm**

* Конфигурация объектов (культивары, рецепты крафта, тайминги роста).
* Серверная валидация некоторых действий (по желанию): «можно ли собирать?», «есть ли семена?».
* (Опц.) серверный прогресс/инвентарь игрока, если нельзя доверять клиенту.
* Экспорт «результатов» в game\_core (через фронт в /attempts).

**Что хранить в бд, примеры**

Если ферма в MVP почти вся клиентская — таблиц может вообще не быть. Но часто полезно иметь крошечный набор:

# apps/game\_farm/models.py

class Crop(models.Model):

slug = models.SlugField(primary\_key=True) # "onion", "carrot"

grow\_time\_s = models.IntegerField(default=30) # время роста (если нужно)

yield\_min = models.IntegerField(default=1)

yield\_max = models.IntegerField(default=3)

class Recipe(models.Model): # если есть крафт

slug = models.SlugField(primary\_key=True) # "seed\_bag"

inputs = models.JSONField(default=dict) # {"onion": 3}

output = models.JSONField(default=dict) # {"seed\_bag": 1}

# Персист игрока (только если нужно держать серверную правду)

class FarmState(models.Model):

user = models.OneToOneField(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE, primary\_key=True)

inventory = models.JSONField(default=dict) # {"onion":5,"carrot":2,"seed\_bag":1}

plots = models.JSONField(default=list) # [{id:"tile\_12", crop:"onion", planted\_at:"...", stage:2}]

updated\_at = models.DateTimeField(auto\_now=True)

* Crop/Recipe — «справочники» правил. Их можно держать и в JSON, но таблицы удобнее редактировать в админке.
* FarmState — включать, только если реально нужен серверный инвентарь/античит.

**Эндпоинты game\_farm (MVP)**

**Вариант A — авторитет клиент (бэку только факты)**

* Нет специальных эндпоинтов.
* Фронт по действию шлёт в game\_core /attempts.
* Плюсы: очень быстро собрать. Минусы: античит слабый.

**Вариант B — лёгкая валидация на сервере (рекомендовано)**

POST /api/farm/validate/harvest

{ "object\_id":"tile\_12", "crop":"onion" }

→ { "ok": true, "yield": 2 } # можно добавить "cooldown\_ms": 1000

* Клиент перед исполнением спрашивает «можно?», сервер сверяет по правилам (и, если есть FarmState, по факту: посажено ли, выросло ли, чьё поле).
* После успеха клиент делает POST /attempts (XP).

**Вариант C — серверный инвентарь/прогресс (чуть сложнее)**

POST /api/farm/harvest

{ "object\_id":"tile\_12", "crop":"onion" }

→ { "ok": true, "yield": 2, "inventory": {"onion": 7, ...} }

* Сервер сам изменяет FarmState и возвращает новый инвентарь.
* Клиент всё равно фиксирует результат в /attempts.

Для MVP: начни с A или B. C подключишь, если понадобится строгий контроль.

**Пример сервисов/правил**

# apps/game\_farm/services.py

from .models import Crop, FarmState

def can\_harvest(user, object\_id, crop\_slug):

# Если нет FarmState — просто проверяем справочник и разрешаем

if not FarmState.objects.filter(user=user).exists():

crop = Crop.objects.filter(slug=crop\_slug).first()

return (crop is not None, {"yield": \_rng\_yield(crop)})

# Если есть FarmState — проверяем грядку и стадию роста

state = FarmState.objects.get(user=user)

plot = \_find\_plot(state.plots, object\_id)

if not plot or plot.get("crop") != crop\_slug or not \_is\_grown(plot):

return (False, {"reason": "not\_ready"})

yld = \_rng\_yield(Crop.objects.get(slug=crop\_slug))

return (True, {"yield": yld})

def apply\_harvest(user, object\_id, crop\_slug, amount):

state = FarmState.objects.select\_for\_update().get(user=user)

# снять урожай с грядки, обновить стадию/пусто

\_update\_plot(state.plots, object\_id, empty=True)

state.inventory[crop\_slug] = state.inventory.get(crop\_slug, 0) + amount

state.save()

return state.inventory`

**Взаимодействие фронта с беком**

* При загрузке мини-игры фронт подгружает:
  + i18n-бандл (/cdn/bundles/farm/<locale>.json),
  + индекс/каталог объектов (/games/farm/index.json или сборка из catalog+placements),
  + (опц.) справочник правил с бэка (GET /api/farm/crops, GET /api/farm/recipes) — чтобы отрисовать UI/время роста.
* На действие игрока:
  + показываем подсказку по i18nKey;
  + при клике «собрать»:
    - (Вариант B) дергаем /api/farm/validate/harvest; если ok, показываем анимацию;
    - после успеха — POST /api/attempts с {game\_slug:"farm", kind:"harvest", success:true, score:yld, meta:{crop, object\_id}};
    - обновляем локальный инвентарь/состояние (или применяем, что прислал сервер в Варианте C).

**Границы модуля**

* game\_farm **не считает XP** — только решает «можно/нельзя», «сколько урожая», «какой инвентарь».
* game\_core агрегирует прогресс, streak, XP и рассылает WS-события.
* lexeme занимается словарём игрока и не знает про урожай/инвентарь.

**Примеры**

**Validate (легкая проверка)**

**Request**

POST /api/farm/validate/harvest

{ "object\_id":"tile\_12", "crop":"onion" }

**Response**

{ "ok": true, "yield": 2 }

**Harvest (сервер применяет стейт)**

**Request**

POST /api/farm/harvest

{ "object\_id":"tile\_12", "crop":"onion" }

**Response**

{

"ok": true,

"yield": 2,

"inventory": {"onion": 7, "carrot": 1}

}

**Attempt (в ядро)**

**Request**

POST /api/attempts

{

"game\_slug": "farm",

"kind": "harvest",

"success": true,

"score": 2,

"meta": { "crop": "onion", "object\_id": "tile\_12" },

"idempotency\_key": "uuid-..."

}

**Вариант MVP (упрощённый):**

* Валидация попытки делается на фронте, бэк принимает факт в /api/attempts.
* Если игре нужен серверный стейт — добавить свой game\_<slug> с минимумом моделей.

**Контракт с ядром:**

* Только POST /api/attempts обязателен. Остальное опционально и локально для игры.

**Позже:** вынести серверные правила, матч-результаты, античит.

**5. ❌ analytics — события/статистика (очень просто)**

**Модель:**

* Event(id, user FK nullable, type:str, payload: JSONB, ts)

**Эндпоинты:**

* POST /api/analytics/events — батч {events:[{type, payload}]} (авторизованные).
* *(опц.)* админ-просмотр или выгрузка.

**Позже:** outbox → BI, агрегаты, приватность/удаление.

**6. ❌ payments — заглушка тарифов/подписок**

**Модели (MVP):**

* Plan(key: 'free'|'pro', title, limits JSON)
* Subscription(user, plan\_key, status:'active'|'none', renews\_at nullable)

**Эндпоинты:**

* GET /api/billing/plan — текущий план пользователя.
* *(опц.)* POST /api/billing/mock-upgrade — заглушка для переключения.

**Позже:** Stripe/CloudPayments webhooks, лимиты по планам, платёжные страницы.

**7. ❌ media — хранение ассетов (S3)**

**Подход (простой и безопасный):**

* Приватный S3 бакет, загрузка через **presigned URL**.

**Эндпоинт:**

* POST /api/media/presign → {key, url, fields} (формирует временную подпись).
* Клиент загружает напрямую в S3; сервер хранит только ключ/метаданные при необходимости.

**Позже:** валидации MIME/размера, обработка аудио, очистка.

**8. ❌ celery — фоновые задачи**

**Кратко**

**MVP задачи:**

* recalc\_leaderboards(period=weekly) — по расписанию.
* increment\_streaks / daily\_maintenance — утренний крон.
* *(опц.)* email-рассылки/уведомления.

**Инфра:**

* Broker/Backend: Redis.
* Запуск воркеров: 1–2 процесса достаточно.

**Позже:** очереди при нагрузке, ретраи, мониторинг (Flower).

**Зачем**

* Выполнять периодические и тяжёлые операции **вне веб-запроса**: пересчёты, ежедневные джобы, рассылки, очистки.
* Разгрузить DRF-вьюхи (быстрый ответ → работа уходит в очередь).

**Инфра (MVP)**

* **Broker/Result backend:** Redis (один экземпляр под очереди и каналы допустим).
* **1 beat + 1–2 worker** процессов достаточно.
* **Формат сериализации:** JSON.
* **TZ:** Europe/Helsinki (Celery timezone), все даты в БД — UTC.

celery.py (ядро):

# backend/celery.py

import os

from celery import Celery

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "backend.settings")

app = Celery("backend")

app.config\_from\_object("django.conf:settings", namespace="CELERY")

app.autodiscover\_tasks() # ищет tasks.py в apps/\*

settings.py (минимум):

CELERY\_BROKER\_URL = "redis://redis:6379/1"

CELERY\_RESULT\_BACKEND = "redis://redis:6379/2"

CELERY\_TIMEZONE = "Europe/Helsinki"

CELERY\_ENABLE\_UTC = True

CELERY\_TASK\_SERIALIZER = "json"

CELERY\_RESULT\_SERIALIZER = "json"

CELERY\_ACCEPT\_CONTENT = ["json"]

CELERY\_BEAT\_SCHEDULE = {} # вынесем ниже

Docker-compose (идея):

web -> gunicorn/uvicorn

worker -> celery -A backend.celery worker -l info

beat -> celery -A backend.celery beat -l info

redis -> broker/result

**Очереди/роутинг (просто)**

* Одна общая очередь default (MVP).
* При росте: analytics, emails, maintenance.

CELERY\_TASK\_ROUTES = {

"analytics.\*": {"queue": "analytics"},

"emails.\*": {"queue": "emails"},

}

**Типовые задачи (MVP)**

**1) Пересчёт лидербордов (еженедельно)**

* Собрать лучшие Attempt за неделю → записать таблицу leaderboard\_weekly.
* Кэшировать результат (Redis) для быстрого GET /leaderboard.

# apps/game\_core/tasks.py

from celery import shared\_task

@shared\_task(name="core.recalc\_leaderboards", max\_retries=3, default\_retry\_delay=60)

def recalc\_leaderboards(period="weekly"):

# 1) выбрать попытки за окно времени

# 2) агрегировать (score/xp)

# 3) сохранить в таблицу/кэш

return {"period": period, "ok": True}

**2) Утреннее обслуживание / streak**

* Фоновая проверка «дыр» (редкая, streak в обычном случае считается онлайном при POST /attempts).
* Очистка устаревших ключей Redis, лёгкая вакуумизация кэшей.

@shared\_task(name="core.daily\_maintenance")

def daily\_maintenance():

# очистки, агрегаты, warmup кэшей и т.п.

return True

**3) (Опц.) Емейлы/уведомления**

* Отправка пачками с ретраями.

@shared\_task(name="emails.send\_digest", autoretry\_for=(Exception,), retry\_backoff=60, retry\_kwargs={"max\_retries":5})

def send\_digest(user\_id):

# render + send

return True

**Расписание (beat)**

from celery.schedules import crontab

CELERY\_BEAT\_SCHEDULE = {

"recalc-leaderboards-weekly": {

"task": "core.recalc\_leaderboards",

"schedule": crontab(minute=0, hour=3, day\_of\_week="mon"), # Пн 03:00

"args": ("weekly",)

},

"daily-maintenance": {

"task": "core.daily\_maintenance",

"schedule": crontab(minute=0, hour=4), # ежедневно 04:00

},

# пример: рассылка — по будням в 9:30

# "emails-digest": { "task": "emails.send\_digest\_batch", "schedule": crontab(minute=30, hour=9, day\_of\_week="1-5") }

}

**Надёжность и идемпотентность**

* **Идемпотентные задачи**: при возможности делайте «не вредно выполнить дважды». Пример: recalc перезаписывает агрегаты, а не инкрементит.
* **Ретраи**: для внешних сервисов (emails) включите autoretry\_for, retry\_backoff.
* **Locks**: для «единственного» выполнения задачи используйте Redis-лок:

import redis, time

r = redis.Redis.from\_url(CELERY\_BROKER\_URL)

def with\_lock(key, ttl=600):

def deco(fn):

def wrap(\*a, \*\*kw):

if not r.set(key, 1, nx=True, ex=ttl): return "locked"

try: return fn(\*a, \*\*kw)

finally: r.delete(key)

return wrap

return deco

**Мониторинг**

* **Flower** для live-мониторинга (flower --broker=redis://...).
* Логи задач (l info) в stdout → собирает ваш логгер/хостинг.
* По мере роста — метрики (Prometheus, celery-exporter).

**Интеграция с остальными модулями**

* game\_core: задачи берут данные из Attempt/UserProfile и пишут в агрегаты/кэш.
* users: ночные рассылки, очистка soft-deleted.
* lexeme: периодические экспорты/импорты словарей (если понадобится).
* channels: Celery может пушить нотификации (через Django-channels) — например, при готовности недельного лидерборда.

**DoD для Celery (MVP)**

* Поднят Redis, настроены worker и beat.
* Работает минимум 2 задачи по расписанию (recalc\_leaderboards, daily\_maintenance).
* Задачи устойчивы к повторному запуску (идемпотентны), есть базовые ретраи.
* Есть команда/скрипт локального запуска задачи (celery -A backend.celery call core.recalc\_leaderboards --args='[\\"weekly\\"]').
* Документация содержит расписание и назначение каждой задачи.

**9. ☑️ channels — WebSocket (чат и прогресс)**

**Кратко**

**Назначение в MVP:**

* Глобальный чат (одна комната) — опционально.
* Пуш progress.updated после POST /api/attempts (xp\_delta, total\_xp).

**Схема:**

* Канал-лейер: Redis.
* Группы: user\_{id} (личные нотификации), chat\_global (если нужен чат).

**Типы сообщений:**

* progress.updated {xp\_delta, total\_xp}
* *(опц.)* chat.message {user, text, ts}

**Позже:** presence/typing, комнатные чаты, игровые события.

**Что и зачем**

* **Личный канал пользователя**: прислать событие progress.updated сразу после успешного POST /api/attempts (xp\_delta, total\_xp, streak).
* **Глобальный чат** — опционально. Если не нужен сейчас, не подключаем.

**Инфра**

* **ASGI-сервер**: uvicorn/daphne.
* **Channel layer**: Redis.
* **Процессы**: веб (ASGI) + воркер Channels (обычно тот же процесс/реплика).

settings.py (суть):

ASGI\_APPLICATION = "backend.asgi.application"

CHANNEL\_LAYERS = { "default": { "BACKEND": "channels\_redis.core.RedisChannelLayer",

"CONFIG": {"hosts": [("redis", 6379)]} } }

asgi.py (минимум):

import os

from django.core.asgi import get\_asgi\_application

from channels.routing import ProtocolTypeRouter, URLRouter

from channels.auth import AuthMiddlewareStack

import apps.channels\_app.routing as ws\_routing

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "backend.settings")

application = ProtocolTypeRouter({

"http": get\_asgi\_application(),

"websocket": AuthMiddlewareStack(URLRouter(ws\_routing.websocket\_urlpatterns)),

})

**Маршрутизация и группы**

* Группа личных уведомлений: user\_{id}.
* (Опц.) глобальный чат: chat\_global.

apps/channels\_app/routing.py:

from django.urls import path

from .consumers import ProgressConsumer # и ChatConsumer, если нужен

websocket\_urlpatterns = [

path("ws/progress/", ProgressConsumer.as\_asgi()),

# path("ws/chat/", ChatConsumer.as\_asgi()),

]

**Аутентификация WS**

* Используем **cookie-сессию** (как у DRF с SessionAuth) → AuthMiddlewareStack подтянет scope["user"].
* Если у тебя JWT — добавь JWT-middleware для WS или прокинь токен в querystring и проверяй в connect.

**Консьюмеры (только прогресс)**

# apps/channels\_app/consumers.py

from channels.generic.websocket import AsyncJsonWebsocketConsumer

class ProgressConsumer(AsyncJsonWebsocketConsumer):

async def connect(self):

user = self.scope["user"]

if not user or not user.is\_authenticated:

return await self.close(code=4401)

self.group = f"user\_{user.id}"

await self.channel\_layer.group\_add(self.group, self.channel\_name)

await self.accept()

async def disconnect(self, code):

if hasattr(self, "group"):

await self.channel\_layer.group\_discard(self.group, self.channel\_name)

# событие из бекенда

async def progress\_updated(self, event):

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

await self.send\_json(event)

Название хэндлера = имя типа с точками → в методе точки заменяются на подчёркивания: progress.updated → progress\_updated.

**Отправка события после /api/attempts**

В вьюхе (после commit/успеха):

from asgiref.sync import async\_to\_sync

from channels.layers import get\_channel\_layer

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

layer = get\_channel\_layer()

async\_to\_sync(layer.group\_send)(

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

{"type": "progress.updated", "xp\_delta": xp\_delta,

"total\_xp": total\_xp, "streak": streak}

)

Вызови send\_progress\_update(...) после того, как сохранил Attempt и обновил UserProfile.

**Клиент (браузер)**

const ws = new WebSocket("wss://api.example.com/ws/progress/");

ws.onmessage = (ev) => {

const msg = JSON.parse(ev.data);

if (msg.type === "progress.updated") {

// обнови UI: прогресс-бар, XP, тост

updateXP(msg.total\_xp);

toast(`+${msg.xp\_delta} XP`);

}

};

// простая авто-переподключалка

ws.onclose = () => setTimeout(() => connectAgain(), 1500);

**Схема сообщений (зафиксировать)**

* Пуш с бэка → клиенту (личная группа):

{ "type": "progress.updated", "xp\_delta": 10, "total\_xp": 330, "streak": 7 }

(Если когда-нибудь нужен чат:)

{ "type": "chat.message", "user": "alice", "text": "hi!", "ts": "2025-09-21T12:00:00Z" }

**Безопасность и стабильность (минимум)**

* **Origin/Host**: настрой CORS/AllowedHosts на уровне WS (reverse-proxy).
* **Rate-limit**: если включишь чат — дросселируй входящие сообщения на консьюмере.
* **Heartbeat**: включи ping/pong на фронте (или доверяй браузеру) и авто-reconnect.
* **Размер**: ограничь максимальный размер входного сообщения (чат) и частоту.
* **Версионирование**: добавь version:1 в сообщения, если ожидаются будущие изменения.

**Что оставить «на потом»**

* Presence/typing (отслеживание в онлайне).
* Комнатные чаты/пати-каналы.
* WS-события из конкретных mini-игр (например, кооп).
* Отдельный worker для WS при росте нагрузки.

**DoD (MVP)**

* ws://…/ws/progress/ подключается только у аутентифицированных.
* После POST /api/attempts клиент получает progress.updated за <1–2 c.
* Сообщение корректно обновляет UI (XP/стрим тостов).
* Стабильный reconnect и отсутствие ошибок при лог-ауте.