Common.AI.* — Работа с AI-сервисами

Пакет плагинов для выполнения запросов к AI-сервисам (LLM, генерация изображений, речь и т.д.). Содержит: * клиентов и запросы к AI-провайдерам; * асинхронную обработку ответов; * таймауты и ошибки; * сохранение и нормализацию результатов. Пакет не содержит логики принятия решений и не является агентным или сценарным слоем. AI здесь — это внешний сервис, а не субъект.

Common.AI.ImageGen — Генерация изображений

Автор: Art Yg
Версия: 1.0

`ImageGen` — универсальный плагин для **асинхронной генерации изображений** через внешний Webhook Processor, с сохранением результата в lead (URL и/или base64) и поддержкой сценарного выхода (success/error/timeout).

Плагин спроектирован так, чтобы работать в **двухфазном режиме**, как `LLMQuery`:

- **PHASE 1** — отправить запрос (async) и выйти из скрипта
- **PHASE 2** — обработать callback-ответ от процессора и продолжить сценарий

Зачем он существует

В Metabot-сценариях нельзя считать генерацию изображения “быстрой синхронной функцией”:

- внешние провайдеры отвечают с задержкой
- ответ может потеряться
- пользователь может продолжать писать, пока мы ждём
- результат может прийти не в том формате (URL vs b64)
- иногда важен **жёсткий контроль ожидания** (timeout), иначе сценарий “повиснет” навсегда

`ImageGen` стандартизирует этот поток:

- отправляет запрос **через Webhook Processor** (Remote transport)
- ждёт callback и распознаёт “это callback или пользовательский ввод”
- сохраняет результат в атрибуты lead
- может уйти в `successScript` (опционально)
- может уйти в `error.script` / `timeout` (если настроены)

Что использует внутри

`ImageGen` — это обёртка, которая опирается на инфраструктурные компоненты:

- **Common.Remote.RemoteApiCall** — транспорт, отправляет запрос в Webhook Processor и включает `asyncResponse`
- **Webhook Processor** — внешний слой, который делает реальный HTTP-запрос к провайдеру и возвращает callback
- **Common.Platform.AsyncFallback** — таймаут/фоллбек-оркестрация (используется внутри `ImageGen` через `timeout`)
- **Common.AI.Prompts** — компонент резолвинга промптов из таблицы и сборки массивов `system/user`, включая ссылки вида `$alias`
- **локальная таблица провайдеров внутри ImageGen** — mapping baseUrl/endpoint/method

Важно: `RemoteApiCall` как транспорт **в принципе поддерживает любых провайдеров**, но текущая версия `ImageGen` по умолчанию заточена под OpenAI Images API.

Поддерживаемые провайдеры

На текущий момент поддержан:

- `openai`

Если вам нужен другой провайдер (например, Replicate, Stability, Midjourney proxy, внутренний сервис) — **свяжитесь с командой**, и мы расширим плагин.

> Примечание по архитектуре: сейчас таблица провайдеров (`PROVIDERS`) находится **внутри плагина**. При необходимости её можно вынести наружу (в конфиг бота/таблицу/отдельный реестр), чтобы вы могли подключать свои провайдеры без изменения кода плагина.

Что сохраняет

В зависимости от настроек:

- `save.urlAttr` — URL изображения (если провайдер вернул url)
- `save.b64Attr` — base64 JSON (`b64_json`), если пришёл именно он
- `save.rawJsonAttr` — сырой payload ответа (для диагностики)

Двухфазный протокол выполнения

PHASE 1 — отправка запроса

Когда `isFirstImmediateCall = true`:

1. Инициализирует timeout-policy через `Common.Platform.AsyncFallback` (если задан `timeout`)
2. Берёт токен из атрибута бота (по `auth.tokenKey`)
3. Собирает итоговый prompt:
   - если задан `prompts` — собирает `system[] + user[]`
   - элементы массива могут быть:
     - inline строка
     - ссылка `$alias` (берётся из таблицы `promptTable` для `agentName`)
   - если в `prompts` используются `$alias`, то **обязательны** `agentName` и `promptTable`
4. Формирует request body под `/images/generations`
5. Отправляет запрос через `RemoteApiCall.send(..., asyncResponse: true)`
6. (опционально) показывает `messages.wait`
7. Возвращает `false` — сценарий “выходит” и ждёт callback

PHASE 2 — обработка callback

Когда `isFirstImmediateCall = false`:

1. Проверяет, что это действительно callback от процессора (`payload.is_async_response`)
2. Если это не callback (пользователь что-то написал):
   - показывает `messages.processing`
   - возвращает `false`
3. Если callback:
   - снимает таймаут-job через `AsyncFallback.unschedule()`
   - парсит `payload.content`
   - извлекает `url` или `b64_json`

- сохраняет в lead
- если включён `requireUrl` и url нет → ошибка

4. Если задан `successScript` — делает `bot.run(successScript)`, иначе возвращается `true`

Конфигурация

Ниже описаны ключевые блоки `ImageGen.run()`.

Provider + Auth

```
provider: "openai",
auth: { tokenKey: "OPENAI_API_KEY" }
```

- `provider` — ключ провайдера (сейчас актуально `openai`)
- `auth.tokenKey` — имя атрибута бота, где лежит API ключ

Prompts (таблица + массивы)

`ImageGen` поддерживает интерфейс промптов “на выроста” — как в LLMQuery: массивы промптов и табличные ссылки.

```
agentName: "orion",
promptTable: "gpt_prompts",

prompts: {
  system: ["$avatar_brief_generator"],
  user: ["...основной запрос..."]
}
```

Принципы:

- `prompts.system` и `prompts.user` — **массивы строк**.
- элементы массива могут быть:
  - обычной строкой (inline prompt)
  - ссылкой вида `$alias` — тогда текст берётся из `promptTable` для указанного `agentName`
- если вы используете `$alias`, то:

- `agentName` **обязателен**
- `promptTable` **обязателен**
- иначе это конфигурационная ошибка (плагин бросает `throw`, чтобы не было “тихих” генераций без системного слоя)

> Примечание: OpenAI Images API принимает один `prompt` (строкой), поэтому `ImageGen` **склеивает массивы** в итоговый текст (обычно `system` + пустая строка + `user`). Это сделано для унификации с LLMQuery и поддержки других провайдеров/форматов в будущем.

Image параметры

```
image: {
  model: "dall-e-3",
  n: 1,
  size: "1024x1792",
  quality: "standard",
  style: "natural",
  background: null,
  response_format: "url"
}
```

Практика по форматам результата:

- **dall-e-2 / dall-e-3**: можно запросить `response_format: "url"` и получить временный URL
- *gpt-image- модели**: часто отдают `b64_json` (URL может не прийти)

---

requireUrl

```
requireUrl: true
```

Если `true`, то отсутствие URL считается ошибкой (даже если пришёл b64).

Это удобно для MVP-потока “дай URL, а скачивание/сохранение сделаю потом”.

Таймаут (fallback)

```
timeout: {
  seconds: 120,
  script: "Orion_Image_Timeout"
}
```

Если callback **не пришёл** за указанное время — планировщик Metabot запускает `timeout.script`.

Таймаут реализован через `Common.Platform.AsyncFallback` **внутри** `ImageGen`.

Namespace для fallback

```
fallback: {
  namespace: "orion_image_reflection"
}
```

Нужен, чтобы несколько асинхронных операций не конфликтовали.

Если не задан, будет auto:

- `imagegen_${code.toLowerCase()}`

UX сообщения

```
messages: {
  wait: "   Генерируем…",
  processing: "⏳ Подожди, ещё формируется…",
  error: "⚠ Не удалось получить изображение"
}
```

- `wait` — показываем при отправке (PHASE 1)
- `processing` — показываем если пользователь пишет во время ожидания (PHASE 2, но это не callback)
- `error` — сообщение по умолчанию при ошибке (если нет `error.script`)

Ошибки

```
error: {
  script: "Orion_Image_Error",
  flagAttr: "orion_image_error",
  reasonAttr: "orion_image_error_reason"
}
```

- `flagAttr` / `reasonAttr` — сохранить состояние ошибки в lead
- `script` — куда перейти при ошибке (опционально)

Успешный выход

```
successScript: "Orion_Image_Ready"
```

Если не указан — `ImageGen.run()` возвращается в ту же точку (`return true`), без перехода.

Таймаут: встроенный vs внешний

В большинстве сценариев таймаут проще и чище задавать **внутри** `ImageGen` через `timeout`, потому что плагин сам использует `Common.Platform.AsyncFallback`.

Внешнее управление таймаутом имеет смысл, если:

- вы хотите один общий timeout на несколько async-операций
- вы централизованно оркестрируете несколько вызовов с одним namespace/policy

---

Примеры использования

Пример 1 — как используется в Orion: system prompt из таблицы + timeout + error + URL (MVP)

```
/**
 * orion_profiling_reflection_image
 *
 * Назначение:
 * - асинхронно сгенерировать вертикальный "Operator Reflection" образ
 * - сохранить URL в лид (скачивание/сохранение — отдельным шагом)
 * - timeout + error внутри ImageGen (как у LLMQuery)
 */

const ImageGen = require("Common.AI.ImageGen");

return ImageGen.run({
  lead,
  isFirstImmediateCall,

  code: "OrionActorImage",

  // Provider/Auth
  provider: "openai",
  auth: { tokenKey: "OPENAI_API_KEY" },

  // Prompts из таблицы + inline
  agentName: "orion",
  promptTable: "gpt_prompts",
  prompts: {
    // системный слой (табличный)
    system: ["$avatar_brief_generator"],

    // основной запрос (inline)
    user: [`
Create a vertical codex-grade mythotech Operator icon in Aurum Void aesthetic.

Aurum Void: cold matte gold schematic lines (axes, nodes, ritual UI glyphs) over deep graphite
```

```
void.
No glossy sci-fi, no neon, no superhero vibe, no humor.

Faceless figure (hood/shadow/mask/void), calm, stable, centered on a strong vertical axis.
Heavy materials: carbon composite armor/robe, matte metal, dense fabric, worn realistic
textures.
Subtle fog/particles for depth.

This is NOT a human portrait. It is an operational manifestation of a system entrepreneur /
product leader at the scaling stage.
Output: a visual artifact, not an illustration.
    `.trim()]
  },

  // Таймаут (fallback)
  timeout: {
    seconds: 120,
    script: "Orion_Image_Timeout"
  },

  // Ошибки (как в LLMQuery)
  error: {
    script: "Orion_Image_Error",
    flagAttr: "orion_image_error",
    reasonAttr: "orion_image_error_reason"
  },

  // Namespace чтобы не конфликтовать
  fallback: {
    namespace: "orion_image_reflection"
  },

  // MVP: хотим именно URL
  requireUrl: true,

  image: {
    model: "dall-e-3",
    size: "1024x1792",
    quality: "standard",
    style: "natural",
```

```
    response_format: "url"
  },

  messages: {
    wait: "   ORION формирует визуальное отражение…",
    processing: "  Подожди, образ ещё куется…",
    error: "⚠  Не удалось получить изображение"
  },

  save: {
    urlAttr: "orion_actor_image_url",
    b64Attr: "orion_actor_image_b64",
    rawJsonAttr: "orion_actor_image_payload"
  }

  // successScript: "Orion_Image_Ready" // опционально
});
```

Пример 2 — минимальный сценарий с таблицей промптов, без successScript

Подходит, когда вы хотите вернуться “в ту же точку” и решать дальше в текущем шаге.

```
const ImageGen = require("Common.AI.ImageGen");

return ImageGen.run({
  lead,
  isFirstImmediateCall,

  code: "OperatorIcon",

  provider: "openai",
  auth: { tokenKey: "OPENAI_API_KEY" },

  agentName: "orion",
  promptTable: "gpt_prompts",
  prompts: {
```

```
    system: ["$avatar_brief_generator"],
    user: ["Create a vertical mythotech Operator icon in Aurum Void aesthetic..."]
  },

  timeout: { seconds: 90, script: "Image_Timeout" },
  error: { script: "Image_Error" },

  requireUrl: true,

  image: {
    model: "dall-e-3",
    size: "1024x1792",
    response_format: "url"
  },

  save: {
    urlAttr: "operator_icon_url",
    rawJsonAttr: "operator_icon_payload"
  }
});
```

Пример 3 — внешний контроль timeout через AsyncFallback (продвинутый режим)

Этот способ полезен, если:

- у вас одна общая политика таймаутов
- вы управляете фоллбеками централизованно (например, несколько разных async-операций в одном шаге)

```
const ImageGen = require("Common.AI.ImageGen");
const AsyncFallback = require("Common.Platform.AsyncFallback");

if (isFirstImmediateCall) {
  AsyncFallback.configure({
    lead,
```

```
    namespace: "orion_image_reflection",
    timeout: { seconds: 120, script: "Orion_Image_Timeout" },
    error: { flagAttr: "orion_image_error", reasonAttr: "orion_image_error_reason" }
  }).schedule();
}

const res = ImageGen.run({
  lead,
  isFirstImmediateCall,

  code: "OrionActorImage",
  provider: "openai",
  auth: { tokenKey: "OPENAI_API_KEY" },

  agentName: "orion",
  promptTable: "gpt_prompts",
  prompts: {
    system: ["$avatar_brief_generator"],
    user: ["Create a vertical mythotech Operator icon in Aurum Void aesthetic..."]
  },

  // timeout внутри не задаём, потому что контролируем снаружи
  error: {
    script: "Orion_Image_Error",
    flagAttr: "orion_image_error",
    reasonAttr: "orion_image_error_reason"
  },

  requireUrl: true,

  image: {
    model: "dall-e-3",
    size: "1024x1792",
    response_format: "url"
  },

  save: {
    urlAttr: "orion_actor_image_url",
    rawJsonAttr: "orion_actor_image_payload"
  }
```

```
});

if (!isFirstImmediateCall && res === true) {
  AsyncFallback.configure({
    lead,
    namespace: "orion_image_reflection"
  }).unschedule();
}

return res;
```

Частые сценарии отказов и как реагировать

- **send_failed** — процессор не принял запрос / RemoteApiCall не отработал → `error.script` или retry
- **provider_error** — провайдер вернул ошибку (например, HTTP != 200) → смотреть `save.rawJsonAttr`, логировать, предлагать retry
- **url_missing** при `requireUrl=true` → либо переключиться на `b64_json`, либо поменять модель/response_format
- **no_result** — ответ пришёл, но данных нет → сохранить `rawJsonAttr` и смотреть, что вернул провайдер/процессор
- **timeout** (через `timeout.script`) → отдельный timeout-script может предложить повторить генерацию или вернуться в меню

Итог

`Common.AI.ImageGen` — стандартизированный способ подключить генерацию изображений в сценарии Metabot:

- двухфазный async flow
- сохранение результата в lead
- UX на случай “пользователь пишет во время ожидания”
- встроенный timeout через `Common.Platform.AsyncFallback`
- работа с промптами как с массивами, включая табличные `$alias` через `Common.AI.Prompts`

- расширяемая система провайдеров (сейчас OpenAI)

Common.AI.Prompts — Универсальный резолвер и сборщик промптов

Автор: Art Yg
Версия: 1.0

`Prompts` — инфраструктурный helper для **унифицированной работы с промптами** в сценариях Metabot и внутри других плагинов (например, `Common.AI.ImageGen`, `LLMQuery/LMClient`).

Он решает типовую боль: **не держать промпты в коде**, не копировать одно и то же “склеивание”, и иметь единый механизм:

- подтянуть промпт из таблицы по ссылке (`$name`, `@name`)
- применить макросы на основе `lead` и `bot`
- нормализовать вход (строка/массив/что угодно)
- собрать итоговый текст промпта из блоков (`system/user/last`)

Зачем он существует

В продуктовых сценариях промпты обычно:

- растут и ветвятся по агентам (`agentName`)
- переезжают в таблицы/консоль управления (а не в JS)
- используют переменные окружения (lead/bot attrs)
- собираются из нескольких частей (“системный каркас” + “задача” + “контекст”)

Без `Prompts` это превращается в дублирование:

- вручную ходим в `table.find(...)`
- вручную проверяем, что `$alias` нашли
- вручную подставляем `lead.getAttr(...)`
- вручную склеиваем массивы

`Prompts` стандартизирует это как **маленький инфраструктурный слой**, который можно подключать где угодно.

Основные принципы

- **Lead-first** — `lead` передаётся явно (без неявной магии).
- **Bot runtime** — `bot` берётся из глобального окружения runtime.
- **Refs only if asked** — в таблицу лезем **только если** ты используешь `$...` или `@...`.
- **Strict by default** — если промпт по ссылке не найден → ошибка (чтобы не генерить “пустоту” молча).
- **Composable** — подходит и как отдельный helper, и как зависимость внутри других плагинов.

Минимальные требования

Для работы в “inline-режиме” достаточно:

1. `lead`
2. вызвать `Prompts.buildText(...)`

Для работы с табличными ссылками (`$...` / `@...`) дополнительно нужно:

1. наличие `table.find` в runtime
2. таблица промптов (`promptTable`)
3. агент (`agentName`) — **обязателен только для** `$name`

Что умеет Prompts

1) Ссылки на промпты из таблицы

Поддерживаются два типа ref:

- `$name` — промпт из таблицы для **конкретного агента**
- `@name` — промпт из таблицы для **общего агента** `<<common>>`

Пример:

- `$avatar_brief_generator` → `agentName = "orion"`
- `@safety_rules` → `agentName` не нужен (используется `<<common>>`)

---

2) Макросы (переменные из lead и bot)

`Prompts` умеет подставлять значения из атрибутов:

- `{{$key}}` → из `lead` (attr или json)
- `{{$$key}}` → из `bot` (attr или json)

Поведение:

- если найден json-атрибут → подставляет `JSON.stringify(value)`
- иначе подставляет строковый `getAttr`
- если не найдено → подставляет пустую строку

---

3) Нормализация входов и сборка блоков

На вход можно дать:

- `null/undefined` → станет `[]`
- `string` → станет `[string]`
- `array` → останется array
- `object/number` → станет `[String(value)]`

Дальше `Prompts` собирает:

- `system[]`
- `user[]`
- `last[]`
- `all[]` = `system + user + last`

И может вернуть:

- либо blocks (`buildBlocks`)
- либо финальную строку (`buildText`) через `join("\n\n")`

Конфигурация

Все методы используют общие опции.

DEFAULTS

```
{
  promptTable: "gpt_prompts",
  agentName: null,   // обязателен для "$name"
  strict: true,      // если промпт не найден → throw
  applyMacros: true
}
```

Важные правила

- `agentName` **обязателен только** если ты используешь `$name`
- `promptTable` обязателен для `$name` и `@name`
- если нет `$` / `@` refs — можно вообще не передавать `agentName/promptTable`

API Prompts

Prompts.toArray(value)

Нормализует значение в массив строк.

Используется внутри, но можно использовать и снаружи.

Prompts.resolveOne(ref, opts)

Резолвит одну строку:

- `$name` → ищет в `promptTable` по `agentName`
- `@name` → ищет в `promptTable` по агенту `<<common>>`
- обычная строка → возвращается как есть

Prompts.resolveMany(list, opts)

Резолвит список refs/строк → массив строк.

Prompts.applyMacros(str, lead)

Применяет:

- `{{$key}}` из lead
- `{{$$key}}` из bot

Prompts.buildBlocks(input, opts)

Собирает блоки по секциям:

```
{
  system: [],
  user: [],
  last: [],
  all: []
}
```

Prompts.buildText(input, opts)

Собирает итоговый prompt как строку:

- вызывает `buildBlocks`
- склеивает `all.join("\n\n")`

Табличный формат промптов

`Prompts` ожидает, что таблица (`promptTable`) содержит хотя бы поля:

- `agent_name`
- `name`
- `prompt`

Запрос выполняется через:

```
table.find(promptTable, ["prompt"], [
  ["agent_name", agent],
  ["name", name]
]);
```

Использование

Ниже примеры именно “для статьи”: чтобы читатель увидел, что есть несколько режимов и зачем это.

Примеры

Пример 1 — Inline: без таблиц, без агента

Подходит для простых сценариев и MVP.

```
const Prompts = require("Common.AI.Prompts");

const prompt = Prompts.buildText(
  {
    system: [
      "You are a strict image generator. Output must be cinematic, realistic, and calm."
    ],
    user: [
      "Create a vertical mythotech Operator icon in Aurum Void aesthetic."
    ]
  },
```

```
  { lead } // agentName/promptTable не нужны
);

// prompt — готовая строка, без обращений к таблицам
```

Пример 2 — Табличный промпт агента: `$alias`

Если используешь `$...`, то **agentName обязателен**, иначе будет throw.

```
const Prompts = require("Common.AI.Prompts");

const prompt = Prompts.buildText(
  {
    system: [
      "$avatar_brief_generator" // берём из таблицы для агента orion
    ],
    user: [
      "Output should be a codex-grade artifact. No neon. No superhero vibe."
    ]
  },
  {
    lead,
    agentName: "orion",
    promptTable: "gpt_prompts"
  }
);
```

Пример 3 — Общий промпт: `@alias` (agentName не нужен)

`@name` всегда читается из агента `<<common>>`.

```
const Prompts = require("Common.AI.Prompts");
```

```
const prompt = Prompts.buildText(
  {
    system: [
      "@safety_rules",
      "@style_aurum_void"
    ],
    user: [
      "Create an abstract Operator icon."
    ]
  },
  {
    lead,
    promptTable: "gpt_prompts"
  }
);
```

Пример 4 — Макросы: подтягиваем контекст из lead и bot

```
const Prompts = require("Common.AI.Prompts");

// допустим:
// lead.getAttr("actor_stage") = "scaling"
// bot.getAttr("BRAND_TONE") = "discipline, meaning, depth"

const prompt = Prompts.buildText(
  {
    system: [
      "Tone: {{$$BRAND_TONE}}"
    ],
    user: [
      "Stage: {{$actor_stage}}",
      "Create a vertical Operator artifact."
    ]
  },
  {
    lead,
```

```
    applyMacros: true
  }
);
```

Пример 5 — Сборка blocks отдельно (для отладки/логирования)

Иногда полезно видеть, какие блоки получились до склейки.

```
const Prompts = require("Common.AI.Prompts");

const blocks = Prompts.buildBlocks(
  {
    system: ["@style_aurum_void", "$avatar_brief_generator"],
    user: ["Generate an icon for current stage: {{$actor_stage}}"]
  },
  {
    lead,
    agentName: "orion",
    promptTable: "gpt_prompts"
  }
);

// blocks.system / blocks.user / blocks.all — можно сохранить в lead или tracer
```

Пример 6 — Мягкий режим (strict=false)

Иногда нужен режим “не падать”, а вернуть сообщение/заглушку.

```
const Prompts = require("Common.AI.Prompts");

const prompt = Prompts.buildText(
  {
    system: ["$missing_alias"],
    user: ["Create an icon."]
  },
```

```
  {
    lead,
    agentName: "orion",
    promptTable: "gpt_prompts",
    strict: false
  }
);

// В strict=false при отсутствии промпта вернётся текст-предупреждение (а не throw)
```

Использование внутри других плагинов

`Common.AI.Prompts` специально сделан как “маленький кусок инфраструктуры”, чтобы:

- не тащить целиком LMClient ради таблиц промптов
- не повторять код резолва в каждом AI-плагине
- унифицировать поведение `$alias` / `@alias` и макросов

Где он уже естественно применяется

- `Common.AI.ImageGen` — чтобы собирать итоговый `image.prompt` из блоков и таблиц
- потенциально любой “AI transport plugin”: STT, TTS, embeddings, classification, etc.

Частые ошибки и как их избежать

1) Использовали `$alias`, но не указали agentName

Это **ошибка по контракту**, будет throw:

- потому что `$alias` — агентный промпт
- без `agentName` невозможно определить неймспейс

Решение: передай `agentName`.

---

2) Использовали `$alias` / `@alias`, но не указали promptTable

Это тоже ошибка:

- потому что непонятно, откуда искать

Решение: передай `promptTable` (или оставь дефолт `"gpt_prompts"`).

---

3) Хотели “просто текст”, но случайно начали строку с `$`

Если текст реально должен начинаться с `$`, то сейчас это будет воспринято как ref.

Практический паттерн: не начинай “сырой текст” с `$` / `@`. Если прям надо — лучше добавить пробел или явную экранировку на уровне твоего контента.

Итог

`Common.AI.Prompts` — это базовый инфраструктурный helper, который:

- даёт единый формат сборки промптов (`system/user/last`)
- вытаскивает промпты из таблиц (`$agent`, `@common`)
- подставляет переменные из lead/bot
- снижает дублирование и делает AI-плагины проще и чище

Он может использоваться:

- **как самостоятельный utility** в сценариях
- **как зависимость** внутри более крупных AI-плагинов (например, `ImageGen`)

Если в твоей архитектуре дальше появятся новые источники промптов (реестр провайдеров, JSON-конфиги, версии промптов, AB-тесты) — вот этот слой и будет правильным местом для расширения. И это как раз тот случай, где можно внезапно сделать себе ловушку, если начать “подмешивать” сюда бизнес-логику — держи его инфраструктурным, иначе потом будет боль.