Common.Platform.* — Платформенные примитивы исполнения

Пакет системных плагинов, формирующих основу исполнения, контекста и жизненного цикла диалогов и процессов внутри платформы.

Этот пакет содержит платформенные примитивы, которые:

* управляют состоянием выполнения,
* хранят и передают контекст,
* используются guard’ами, input-компонентами и системными проверками,
* не зависят от конкретных сценариев, каналов или бизнес-доменов.

Platform не является:
— бизнес-логикой,
— сценарным фреймворком,
— workflow-движком,
— пользовательским интерфейсом,
— конкретным каналом ввода (voice / text / ui).

Platform — это инфраструктурный слой, на котором строятся остальные механизмы системы.

- Common.Platform.AsyncFallback — Универсальный helper для таймаутов асинхронных операций
- Common.Platform.DeepLinks — входы из рекламных кампаний и маршрутизация

Common.Platform.AsyncFallback — Универсальный helper для таймаутов асинхронных операций

Автор: Art Yg
Версия: 1.0

`AsyncFallback` — это платформенный helper, который решает одну практическую проблему: **как безопасно обработать ситуацию “мы отправили async-запрос, но ответ может не прийти вовремя или вообще не прийти”**.

Он нужен для любых операций, где есть “PHASE 1 → отправили запрос” и “PHASE 2 → пришёл callback”:

- LLM (Remote LLM Query)
- ImageGen (генерация изображений)
- STT / Voice Transcription
- любые RemoteApiCall (внешние API, webhook processor, интеграции)

`AsyncFallback` **не является AI-модулем**. Это оркестрация платформенного уровня.

Зачем он существует

В реальном мире async-вызовы ломаются не потому что “код плохой”, а потому что:

- провайдер завис / долго отвечает
- callback потерялся на транспорте
- webhook processor упал
- ответ пришёл, но неполный (нарушение контракта)
- пользователь продолжает писать, пока операция ещё не завершилась

Без таймаута сценарий может **залипнуть**: пользователь пишет, а система “ждёт” бесконечно.

`AsyncFallback` решает это через стандартный механизм Metabot Scheduler:

1. **Планирует job**: “через N секунд выполнить fallback-script”
2. **Отменяет job**, когда callback успешно пришёл
3. Позволяет фиксировать **причину ошибки** и **детали**, чтобы дальше можно было принять решение (редирект, retry, сообщение пользователю)

Основные принципы

- **Platform-layer** — не привязан к AI, применяется везде
- **Namespace-based** — несколько параллельных async-операций не конфликтуют
- **Opt-in** — если timeout не задан, ничего не планируется
- **Side-effect only** — не вмешивается в бизнес-логику, только планирует/снимает job и пишет маркеры
- **Debug-friendly** — сохраняет конфиг и last_reason/last_details в lead (по namespace)

Минимальные требования

Чтобы использовать `AsyncFallback`, нужно:

1. Иметь доступ к **планировщику** Metabot:
   - `bot.scheduleJob({ lead_id, script_code, run_after_sec })`
   - `bot.clearJobsByScriptCode(script_code, lead_id)`
2. Иметь `leadId` (или возможность вычислить lead_id)

`AsyncFallback` не требует базы данных, таблиц или дополнительных сервисов.

Концепция Namespace

Namespace — обязательный “scope” для идентификации конкретной операции.

Это важно, потому что на одном lead могут одновременно идти:

- ImageGen (ожидаем картинку)

- LLMQuery (ожидаем JSON)
- STT (ожидаем текст транскрипции)

Если бы мы хранили “timeout.script” без namespace — они бы перетирали друг друга.

Пример namespace:

- `orion_image_reflection`
- `llm_actor_profile_extract`
- `voice_transcription_q1`

Конфигурация

`AsyncFallback` конфигурируется на вызове:

- `namespace` — обязательно
- `timeout` — опционально
- `error` — опционально
- `storageRoot` — почти всегда не трогаем

Поля конфигурации

- `timeout.seconds` — через сколько секунд считать операцию “просроченной”
- `timeout.script` — какой скрипт выполнить по истечении таймаута
- `error.flagAttr` — атрибут флага ошибки на lead (например `true/false`)
- `error.reasonAttr` — атрибут причины ошибки (строка)

Как использовать

PHASE 1 — отправка async-запроса (isFirstImmediateCall)

1. Конфигурируем fallback
2. Планируем таймаут
3. Отправляем remote request
4. Возвращаем `false` (ждём callback)

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

const fb = AsyncFallback.configure({
  lead,
  namespace: "orion_image_reflection",
  timeout: { seconds: 120, script: "Orion_Image_Timeout" },
  error: { flagAttr: "orion_image_error", reasonAttr: "orion_image_error_reason" }
});

fb.schedule();

// ... RemoteApiCall.send(..., asyncResponse: true)
return false;
```

PHASE 2 — пришёл callback

1. Снимаем таймаут (если он был)
2. Валидируем результат
3. Если контракт нарушен — `fail(reason, details)`
4. Дальше ты решаешь: редиректить в error-script или вернуться “в ту же точку”

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

const fb = AsyncFallback.configure({
  lead,
  namespace: "orion_image_reflection"
});

fb.unschedule();

// если ответ плохой (например нет url при requireUrl=true)
fb.fail("url_missing", { requireUrl: true });

// твоя стратегия выхода:
return bot.run({ script_code: "Orion_Image_Error" });
```

Методы

`AsyncFallback.configure(params) → instance`

Создаёт instance и **сохраняет конфиг** в lead под namespace.
Главная точка входа. Используй и в PHASE 1, и в PHASE 2 — единообразно.

`instance.schedule() → boolean`

Планирует fallback-job, если `timeout.seconds` и `timeout.script` заданы.

Поведение:

- если timeout не задан → возвращает `false`, не считается ошибкой
- перед планированием **снимает предыдущие jobs** этого script_code для lead (защита от дублей)
- выставляет служебный флаг `active=1`

`instance.unschedule() → boolean`

Отменяет запланированный fallback-job (если он был).

Поведение:

- если script неизвестен → `false`
- снимает `active=0` даже если clearJobs вернул false (чтобы состояние не “залипало”)

`instance.fail(reason, details?) → true`

Фиксирует ошибку и причину:

- `error.flagAttr = true` (если задан)
- `error.reasonAttr = reason` (если задан)
- дополнительно пишет namespace-атрибуты:
  - `last_reason`
  - `last_details` (JSON/string)

Это **не редирект** и **не exception** — это маркер, после которого ты сам решаешь, что делать.

`instance.clear() → true`

Очищает служебные данные namespace:

- config
- script/seconds/active
- last_reason/last_details

Полезно после успешного завершения операции (опционально).

Типовой паттерн: “пользователь пишет, пока ждём”

`AsyncFallback` — про таймаут, но он закрывает важный кусок UX:

- если пользователь продолжает писать, пока async-операция ещё ждёт callback, ты показываешь “ ждите…”
- если callback так и не пришёл — fallback-job переведёт сценарий в timeout-script

Обычно это делается так:

- **проверяешь** `payload.is_async_response`
- если это не callback — отправляешь `processing` и возвращаешь `false`
- callback → `unschedule()`

(Эта логика живёт в конкретном плагине типа ImageGen/LLMQuery, а AsyncFallback даёт им общий таймаутный механизм.)

Когда использовать

- есть async callback и риск “зависнуть”
- важно гарантировать, что сценарий не будет ждать бесконечно
- нужен единый механизм таймаута для разных компонентов
- хочешь стандартизировать “timeout-script” как часть контракта компонента

Когда не использовать

- операция строго синхронная
- у операции нет понятного “deadline” (таймаут не имеет смысла)
- ты уже используешь другой механизм оркестрации таймаутов, и второй будет конфликтовать

Практические замечания, чтобы не словить редкий ад

Поздний callback после таймаута

Может случиться: таймаут-скрипт уже отработал, а потом всё же прилетел success-callback. `AsyncFallback` снимает job на callback, но **если job уже выполнился**, снять уже нечего.

Правильный паттерн на уровне компонента:

- timeout-script ставит флаг `*_timed_out = true`
- callback-обработчик проверяет флаг и **игнорирует поздний успех** (или делает компенсацию)

Это не обязанность `AsyncFallback`, потому что стратегия зависит от бизнес-логики.

Итог

`Common.Platform.AsyncFallback` — простой, но критически полезный слой платформенной оркестрации:

- даёт **гарантию выхода** из ожидания
- позволяет вести **несколько async-операций параллельно**
- стандартизирует **timeout и error markers**
- не превращается в “монолитный менеджер всего” — остаётся лёгким helper’ом

Common.Platform.DeepLinks — входы из рекламных кампаний и маршрутизация

Пакет: `Platform`

Полное имя компонента: `Common.Platform.DeepLinks`

Что это

`DeepLinks` — это платформенный компонент Metabot для обработки входов по deeplink-ссылкам.

Он принимает параметры из внешней ссылки, нормализует их, сохраняет контекст входа в атрибуты лида и вычисляет сценарный маршрут, по которому бот должен повести пользователя дальше.

Проще говоря:

```
DeepLinks — это входной шлюз между рекламой, лендингом, рассылкой, QR-кодом или внешней точкой входа и логикой сценария внутри бота.
```

Компонент особенно нужен там, где важно понимать:

```
откуда пришёл человек
по какой гипотезе
с какого оффера
с какого лендинга
с какой кампании
с какого креатива
```

```
в какую воронку его вести
какой сценарный route запустить
```

Такой стиль документации повторяет подход, который уже используется в компонентах `LLMQuery` и `VoiceInput`: сначала описывается назначение компонента, затем сценарии применения, схема работы, параметры и примеры вызова.

Зачем нужен DeepLinks

Без стандартизированной обработки deeplink-входов рекламные тесты быстро превращаются в хаос.

Через месяц становится непонятно:

```
какая гипотеза дала лида
какой баннер сработал
какой лендинг привёл диагностику
какой канал дал созвон
какой route запустился
какой источник был первым
какой источник был последним
```

`DeepLinks` решает эту задачу на уровне платформы.

Он превращает внешний переход в структурированный entry context:

```
deeplink params
→ normalized entry context
→ lead attributes
→ route
→ сценарная ветка
→ диагностика / консультация / sales handoff
```

Где используется

Компонент подходит для:

```
рекламных кампаний
лендингов
UTM-меток
Telegram / MAX / VK deeplink-входов
рассылок по старой базе
QR-кодов
контентных кампаний
A/B-тестов
Demand Lab
партнёрских программ
мероприятий
внешних виджетов
```

Типовой кейс:

```
Пользователь видит рекламу
→ переходит на лендинг
→ нажимает кнопку Telegram / MAX
→ попадает в бота по deeplink
→ DeepLinks фиксирует источник и параметры
→ бот запускает нужную диагностику
```

Где находится компонент

Компонент находится в пакете **Platform** и подключается как обычный плагин Metabot:

```
const DeepLinks = require("Common.Platform.DeepLinks")
```

Основной метод:

```
DeepLinks.handle({...})
```

Как работает DeepLinks

`DeepLinks` работает как синхронный входной компонент.

Он не делает внешних API-запросов и не ждёт callback. В отличие от `LLMQuery`, которому нужен двухфазный async-процесс, и `VoiceInput`, у которого есть многофазный pipeline, `DeepLinks` выполняется сразу внутри обычной команды `Run JavaScript`. Для сравнения: `LLMQuery` специально работает через `Run asynchronous API-request`, потому что ждёт внешний LLM callback, а `VoiceInput` проходит через processor script и STT callback.

Общая схема

```
Run JavaScript
↓
DeepLinks.handle({ lead, ...config })
↓
читает sys_last_script_request_params.deeplink
↓
нормализует параметры
↓
нормализует action
↓
вычисляет route
↓
сохраняет __entry_*
↓
сохраняет __first_entry_*
↓
сохраняет business aliases, например demand_*
↓
сохраняет __last_deeplink_entry_summary
↓
при необходимости отправляет Telegram-уведомление
↓
возвращает entry context
```

После этого сценарий использует `entry.route` или атрибут `__entry_route` в условиях MenuBuilder.

Что делает метод `handle`

Базовый вызов:

```
const DeepLinks = require("Common.Platform.DeepLinks")

const entry = DeepLinks.handle({
  lead,
  aliases: {},
  actionAliases: {},
  routes: {},
  notify: {}
})

lead.setAttr("__entry_route", entry.route)
```

Метод делает несколько операций.

1. Читает входные параметры

По умолчанию параметры берутся из:

```
lead.getJsonAttr("sys_last_script_request_params").deeplink
```

То есть компонент работает с тем, что пришло в бот через deeplink.

Пример входа:

```
?a=diag&h=HYP-0001&o=OFF-0001&c=CAMP-0001&l=LP-0001&f=FUN-0001&x=yandex&m=cpc
```

---

2. Нормализует параметры

Компонент поддерживает короткие параметры и алиасы.

Например, все эти варианты могут быть приведены к одному полю `campaignId`:

```
c
cmp
camp
campaign
```

```
utm_campaign
```

То есть ссылка может прийти так:

```
?c=CAMP-0001
```

или так:

```
?utm_campaign=CAMP-0001
```

Внутри компонента это станет:

```
entry.campaignId = "CAMP-0001"
```

---

3. Нормализует action

`action` — это намерение входа.

Например:

```
diag
ask
call
offer
start
```

В ссылке могут прийти разные варианты:

```
?a=diagnostic
?a=voice_diag
?a=diag
```

Через `actionAliases` они приводятся к одному каноническому значению:

```
action = diag
```

Важно: **отдельного объекта `actions` нет**. Канонические actions задаются ключами объекта `actionAliases`.

Пример:

```
actionAliases: {
  diag: ["diag", "diagnostic", "voice", "voice_diag"],
  ask: ["ask", "kb", "knowledge", "consult"],
  call: ["call", "book", "book_call", "meeting"],
  offer: ["offer", "show_offer", "intro"],
  start: ["start", "begin"]
}
```

Чтобы добавить новый action, нужно добавить новый ключ:

```
actionAliases: {
  demo: ["demo", "show_demo", "presentation"]
}
```

После этого ссылка:

```
?a=show_demo
```

будет нормализована в:

```
action = demo
```

---

4. Вычисляет route

`route` — это сценарный маршрут, куда бот должен повести пользователя.

Например:

```
demand_voice_diagnostic
knowledge_consultant
sales_handoff
offer_intro
fallback
```

`DeepLinks` сам не запускает сценарий. Он только вычисляет route и сохраняет его.

Дальше сценарист использует route в условиях MenuBuilder:

```
return lead.getAttr("__entry_route") === "demand_voice_diagnostic"
```

Action и Route

Это важное различие.

Action

`action` описывает, **что хотел сделать пользователь или ссылка**.

Примеры:

```
diag — пройти диагностику
ask — задать вопрос / перейти к консультанту
call — записаться на созвон
offer — посмотреть оффер
start — обычный старт
```

Route

`route` описывает, **куда внутри сценария должен перейти бот**.

Примеры:

```
demand_voice_diagnostic
knowledge_consultant
sales_handoff
offer_intro
fallback
```

Коротко:

```
action = намерение входа
route = технический маршрут сценария
```

Как определяется route

Порядок выбора route:

```
1. Явный route из ссылки: rt / route
2. Route по funnel: f / funnel
3. Route по action
4. defaultDemandRoute, если есть параметры гипотезы / оффера / кампании
5. fallbackRoute
```

1. Явный route из ссылки

Если в ссылке передан `rt`, он имеет максимальный приоритет:

```
?rt=sales_handoff
```

Тогда:

```
route = sales_handoff
```

Это удобно, если дизайнер сценария хочет сам явно указать маршрут.

2. Route по funnel

Если передан funnel:

```
?f=FUN-0001
```

и в конфигурации есть:

```
routes: {
  byFunnel: {
    "FUN-0001": "demand_voice_diagnostic"
  }
}
```

то компонент вернёт:

```
route = demand_voice_diagnostic
```

3. Route по action

Если передан action:

```
?a=diag
```

и в конфигурации есть:

```
routes: {
  byAction: {
    diag: "demand_voice_diagnostic"
  }
}
```

то компонент вернёт:

```
route = demand_voice_diagnostic
```

4. Default route

Если в ссылке есть параметры Demand Lab, но конкретный route не найден:

```
h / o / c / l / e / s
```

компонент может отправить пользователя в route по умолчанию:

```
defaultDemandRoute: "demand_voice_diagnostic"
```

5. Fallback

Если ничего не подошло:

```
fallbackRoute: "fallback"
```

Конфигурация компонента

`DeepLinks` можно использовать почти без конфигурации, но в боевых проектах рекомендуется явно передавать настройки.

Основные блоки:

```
DeepLinks.handle({
  lead,

  attrPrefix: "__entry",
  firstAttrPrefix: "__first_entry",
  businessPrefix: "demand",

  defaultAction: "diag",

  aliases: {},
  actionAliases: {},
  routes: {},
  notify: {}
})
```

Что параметризовано

В компоненте параметризуются:

```
какие входные параметры считать синонимами
какие actions существуют
какие алиасы есть у actions
какой route соответствует action
какой route соответствует funnel
куда сохранять technical attrs
куда сохранять first entry
какой business prefix использовать
```

```
отправлять ли уведомление в Telegram
какие bot attrs использовать для Telegram-уведомления
```

То есть компонент не привязан жёстко к Demand Lab. Demand Lab — это одна из возможных конфигураций.

Почему не надо зашивать всё в сценарий

Без плагина каждый сценарий будет заново писать одно и то же:

```
pick()
normalize params
UTM aliases
first entry
last entry
route resolution
support notification
summary JSON
```

Это быстро приведёт к разным стандартам в разных ботах.

`Common.Platform.DeepLinks` нужен, чтобы стандартизировать входы на уровне платформы, а не плодить несовместимые копии кода.

Что компонент не делает

`DeepLinks` не заменяет сценарную логику.

Он не делает:

```
не создаёт рекламные кампании
не проверяет, есть ли HYP/OFF/CAMP в Excel
не запускает VoiceInput
```

```
не анализирует ответы пользователя
не делает lead scoring
не сохраняет историю всех входов в отдельную таблицу
не заменяет CRM или аналитику
```

Его задача уже:

```
принять вход
нормализовать
сохранить
вычислить route
дать сценарию точку продолжения
```

Историю всех входов позже лучше выносить в отдельную таблицу, например:

```
demand_entry_events
```

Кратко

```
DeepLinks — это платформенный компонент входа.

Он читает deeplink-параметры, приводит их к единому формату, сохраняет first/last entry
context, вычисляет route и возвращает entry context сценарию.

Actions задаются через actionAliases.
Routes задаются через routes.byAction / routes.byFunnel / defaultDemandRoute / fallbackRoute.

Компонент не запускает сценарий сам.
Он только сохраняет route, который дальше используется в MenuBuilder.
```

Сигнатура, конфигурация и параметры входа

Подключение компонента

Компонент подключается как обычный платформенный плагин:

```
const DeepLinks = require("Common.Platform.DeepLinks")
```

Основной метод:

```
const entry = DeepLinks.handle({...})
```

Метод возвращает объект `entry` — нормализованный контекст входа.

Главное поле:

```
entry.route
```

Именно его дальше используют в условиях MenuBuilder.

Сигнатура `DeepLinks.handle()`

Минимальный вызов:

```
const DeepLinks = require("Common.Platform.DeepLinks")

const entry = DeepLinks.handle({
  lead
})

lead.setAttr("__entry_route", entry.route)
```

Но в боевых проектах лучше передавать конфигурацию явно:

```
const DeepLinks = require("Common.Platform.DeepLinks")

const entry = DeepLinks.handle({
```

```
  lead,

  attrPrefix: "__entry",
  firstAttrPrefix: "__first_entry",
  businessPrefix: "demand",

  defaultAction: "diag",

  aliases: {},
  actionAliases: {},
  routes: {},
  notify: {}
})

lead.setAttr("__entry_route", entry.route)
```

Параметры `handle()`

Параметр	Тип	Обязателен	Описание
`lead`	object	Да	Объект лида Metabot
`raw`	object	Нет	Можно передать deeplink-параметры вручную. Если не передано, компонент берёт их из `sys_last_script_request_params.deeplink`
`attrPrefix`	string	Нет	Префикс технических атрибутов последнего входа. По умолчанию `__entry`
`firstAttrPrefix`	string	Нет	Префикс атрибутов первого входа. По умолчанию `__first_entry`
`businessPrefix`	string	Нет	Префикс бизнес-атрибутов, например `demand`
`saveFirstEntry`	boolean	Нет	Сохранять ли первый вход. По умолчанию `true`
`saveBusinessAliases`	boolean	Нет	Сохранять ли бизнес-атрибуты с `businessPrefix`

Параметр	Тип	Обязателен	Описание
`saveLegacyAliases`	boolean	Нет	Сохранять ли совместимость со старыми `last_entry_*`
`defaultAction`	string	Нет	Action по умолчанию, если в ссылке action не передан
`aliases`	object	Нет	Алиасы входных параметров
`actionAliases`	object	Нет	Справочник канонических actions и их алиасов
`routes`	object	Нет	Правила вычисления route
`notify`	object	Нет	Настройки Telegram-уведомления

Полный пример для Demand Lab

```
const DeepLinks = require("Common.Platform.DeepLinks")

const entry = DeepLinks.handle({
  lead,

  // ========================
  // ATTRIBUTES
  // ========================
  attrPrefix: "__entry",
  firstAttrPrefix: "__first_entry",
  businessPrefix: "demand",

  saveFirstEntry: true,
  saveBusinessAliases: true,
  saveLegacyAliases: true,

  // ========================
  // DEFAULT ACTION
```

```
// ==========================
defaultAction: "diag",

// ==========================
// INPUT PARAMETER ALIASES
// ==========================
aliases: {
  hypothesisId: [
    "h",
    "hyp",
    "hypothesis",
    "hypothesis_id",
    "hid"
  ],

  segment: [
    "s",
    "seg",
    "segment",
    "segment_id",
    "audience",
    "target_segment"
  ],

  offerId: [
    "o",
    "offer",
    "offer_id",
    "oid"
  ],

  landingId: [
    "l",
    "lp",
    "landing",
    "landing_id",
    "page"
  ],

  entryPointId: [
```

```
    "e",
    "ep",
    "entry_point",
    "entry_point_id"
  ],

  campaignId: [
    "c",
    "cmp",
    "camp",
    "campaign",
    "campaign_id",
    "utm_campaign"
  ],

  creativeId: [
    "k",
    "cr",
    "creative",
    "creative_id",
    "utm_content"
  ],

  funnelId: [
    "f",
    "flow",
    "funnel",
    "funnel_id",
    "entry"
  ],

  action: [
    "a",
    "act",
    "action"
  ],

  route: [
    "rt",
    "route"
```

```
    ],

    ref: [
      "r",
      "ref",
      "link_id",
      "entry_ref"
    ],

    source: [
      "x",
      "src",
      "source",
      "utm_source",
      "ref_source"
    ],

    medium: [
      "m",
      "med",
      "medium",
      "utm_medium",
      "ref_medium"
    ],

    term: [
      "t",
      "term",
      "utm_term",
      "keyword",
      "keyword_id"
    ],

    variant: [
      "v",
      "variant",
      "ab",
      "ab_variant"
    ]
  },
```

```
// ========================
// ACTIONS
// ========================
actionAliases: {
  diag: [
    "diag",
    "diagnostic",
    "voice",
    "voice_diag",
    "voice_diagnostic",
    "demand_diag"
  ],

  ask: [
    "ask",
    "kb",
    "knowledge",
    "consult",
    "consultant"
  ],

  call: [
    "call",
    "book",
    "book_call",
    "meeting"
  ],

  offer: [
    "offer",
    "show_offer",
    "intro"
  ],

  demo: [
    "demo",
    "show_demo",
    "presentation"
  ],
```

```
  start: [
    "start",
    "begin"
  ]
},

// ==========================
// ROUTES
// ==========================
routes: {
  byAction: {
    diag: "demand_voice_diagnostic",
    ask: "knowledge_consultant",
    call: "sales_handoff",
    offer: "offer_intro",
    demo: "demo_intro",
    start: "fallback"
  },

  byFunnel: {
    "FUN-0001": "demand_voice_diagnostic",
    "FUN-0002": "knowledge_consultant",
    "FUN-0003": "sales_handoff",
    "FUN-0004": "demo_intro"
  },

  defaultDemandRoute: "demand_voice_diagnostic",
  fallbackRoute: "fallback"
},

// ==========================
// NOTIFICATION
// ==========================
notify: {
  enabled: true,
  tokenAttr: "SUPPORT_TELEGRAM_BOT_TOKEN",
  chatIdAttr: "SUPPORT_TELEGRAM_CHAT_ID",
  title: "□□Новый вход Demand Lab"
}
```

```
})

// Явно сохраняем route для условий MenuBuilder
lead.setAttr("__entry_route", entry.route)
```

Входные параметры deeplink

Компонент поддерживает короткие параметры и длинные алиасы.

Канонические короткие параметры

Параметр	Поле внутри `entry`	Значение
`h`	`hypothesisId`	Гипотеза
`s`	`segment`	Сегмент / ниша
`o`	`offerId`	Оффер
`l`	`landingId`	Лендинг
`e`	`entryPointId`	Точка входа
`c`	`campaignId`	Кампания
`k`	`creativeId`	Креатив / баннер
`f`	`funnelId`	Воронка
`a`	`originalAction` → `action`	Действие / намерение
`rt`	`explicitRoute` → `route`	Явный маршрут
`r`	`ref`	Тип входа / ref
`x`	`source`	Источник
`m`	`medium`	Medium
`t`	`term`	Ключевая фраза
`v`	`variant`	A/B-вариант

Пример deeplink

```
?a=diag&h=HYP-0001&s=S03&o=OFF-0001&c=CAMP-0001&l=LP-0001&f=FUN-
0001&x=yandex&m=cpc&t=obuchenie_partnerov&k=CR-0001&v=A
```

Расшифровка:

Параметр	Значение	Что означает
`a=diag`	`diag`	Пользователь идёт в диагностику
`h=HYP-0001`	`HYP-0001`	Гипотеза
`s=S03`	`S03`	Сегмент
`o=OFF-0001`	`OFF-0001`	Оффер
`c=CAMP-0001`	`CAMP-0001`	Рекламная кампания
`l=LP-0001`	`LP-0001`	Лендинг
`f=FUN-0001`	`FUN-0001`	Воронка
`x=yandex`	`yandex`	Источник
`m=cpc`	`cpc`	Тип трафика
`t=obuchenie_partnerov`	`obuchenie_partnerov`	Ключевая фраза
`k=CR-0001`	`CR-0001`	Креатив
`v=A`	`A`	Вариант теста

Пример с UTM-метками

Можно использовать стандартные UTM:

```
?utm_source=yandex&utm_medium=cpc&utm_campaign=CAMP-0001&utm_content=CR-
0001&utm_term=obuchenie_partnerov
```

Компонент нормализует их так:

UTM	Поле внутри `entry`
`utm_source`	`source`

UTM	Поле внутри `entry`
`utm_medium`	`medium`
`utm_campaign`	`campaignId`
`utm_content`	`creativeId`
`utm_term`	`term`

То есть можно передавать как короткие параметры, так и обычные UTM.

Как добавлять новые actions

Actions задаются через `actionAliases`.

Ключ объекта — это канонический action.

Массив — это список вариантов, которые могут прийти в ссылке.

Пример:

```
actionAliases: {
  audit: [
    "audit",
    "path_audit",
    "partner_audit",
    "check_path"
  ]
}
```

Теперь все эти ссылки:

```
?a=audit
?a=path_audit
?a=partner_audit
?a=check_path
```

будут нормализованы в:

```
action = audit
```

Чтобы этот action вёл в нужный сценарий, нужно добавить route:

```
routes: {
  byAction: {
    audit: "partner_path_audit"
  }
}
```

И дальше в MenuBuilder:

```
return lead.getAttr("__entry_route") === "partner_path_audit"
```

Как добавлять новые routes

Routes задаются в блоке `routes`.

Route по action

```
routes: {
  byAction: {
    diag: "demand_voice_diagnostic",
    call: "sales_handoff",
    demo: "demo_intro"
  }
}
```

Если ссылка:

```
?a=demo
```

то route будет:

```
demo_intro
```

Route по funnel

```
routes: {
  byFunnel: {
    "FUN-0001": "demand_voice_diagnostic",
    "FUN-0002": "knowledge_consultant"
  }
}
```

Если ссылка:

```
?f=FUN-0002
```

то route будет:

```
knowledge_consultant
```

Явный route из ссылки

```
?rt=sales_handoff
```

В этом случае route будет:

```
sales_handoff
```

`rt` имеет самый высокий приоритет.

Что возвращает `handle()`

Метод возвращает объект `entry`.

Пример:

```
{
  "type": "deeplink",
  "ts": "2026-05-09T12:00:00.000Z",

  "hypothesisId": "HYP-0001",
  "segment": "S03",
  "offerId": "OFF-0001",
  "landingId": "LP-0001",
  "entryPointId": "",
  "campaignId": "CAMP-0001",
  "creativeId": "CR-0001",
  "funnelId": "FUN-0001",
  "variant": "A",

  "source": "yandex",
  "medium": "cpc",
  "ref": "",
  "term": "obuchenie_partnerov",

  "originalAction": "diag",
  "action": "diag",
  "explicitRoute": "",
  "route": "demand_voice_diagnostic"
}
```

Этот объект можно использовать сразу в сценарии:

```
if (entry.route === "demand_voice_diagnostic") {
  // дополнительная логика, если нужна
}
```

Но обычно достаточно сохранить:

```
lead.setAttr("__entry_route", entry.route)
```

и дальше использовать route в условиях MenuBuilder.

Пример условий MenuBuilder

Диагностика

```
return lead.getAttr("__entry_route") === "demand_voice_diagnostic"
```

Консультация по базе знаний

```
return lead.getAttr("__entry_route") === "knowledge_consultant"
```

Передача в продажи

```
return lead.getAttr("__entry_route") === "sales_handoff"
```

Показ оффера

```
return lead.getAttr("__entry_route") === "offer_intro"
```

Fallback

```
return lead.getAttr("__entry_route") === "fallback"
```

Практическое правило

Для рекламных кампаний лучше использовать короткие параметры:

```
h, s, o, c, l, f, x, m, t, k, v
```

Для совместимости с рекламными системами можно использовать UTM:

```
utm_source, utm_medium, utm_campaign, utm_content, utm_term
```

Для явного сценарного перехода можно использовать:

```
rt
```

Но `rt` лучше использовать аккуратно. Если все ссылки будут напрямую задавать route, можно потерять управляемость через `funnelId` и `action`.

Оптимальный паттерн:

```
обычный рекламный вход → a + h/o/c/l/f
сложный сценарный вход → rt
совместимость с рекламой → utm_*
```

Что сохраняется в lead, уведомления и отладка

Что сохраняет компонент

После вызова:

```
const entry = DeepLinks.handle({ lead, ...config })
```

компонент сохраняет несколько слоёв данных:

```
1. технический контекст последнего входа: __entry_*
2. технический контекст первого входа: __first_entry_*
3. бизнес-поля для фильтрации: demand_*
4. legacy-поля совместимости: last_entry_*
5. compact summary: __last_deeplink_entry_summary
```

---

1. Последний вход: `__entry_*`

`__entry_*` — это технический слой последнего deeplink-входа.

Он **перезаписывается при каждом новом входе**.

Атрибут	Что хранит
`__entry_type`	Тип входа, обычно `deeplink`
`__entry_ts`	Время входа
`__entry_action`	Нормализованный action
`__entry_original_action`	Action как пришёл в ссылке
`__entry_route`	Вычисленный route
`__entry_hypothesis_id`	Гипотеза
`__entry_segment`	Сегмент
`__entry_offer_id`	Оффер
`__entry_landing_id`	Лендинг
`__entry_entry_point_id`	Точка входа
`__entry_campaign_id`	Кампания
`__entry_creative_id`	Креатив
`__entry_funnel_id`	Воронка
`__entry_variant`	A/B-вариант
`__entry_ref`	Ref / тип входа
`__entry_src`	Источник
`__entry_med`	Medium
`__entry_term`	Ключевая фраза

Атрибут	Что хранит
`__entry_payload_json`	Сырые deeplink-параметры

Пример:

```
__entry_hypothesis_id = HYP-0001
__entry_offer_id = OFF-0001
__entry_campaign_id = CAMP-0001
__entry_route = demand_voice_diagnostic
```

---

2. Первый вход: `__first_entry_*`

`__first_entry_*` нужен, чтобы не потерять первоисточник лида.

Он записывается **только один раз**, если ещё не был заполнен.

Пример:

```
__first_entry_src = yandex
__first_entry_med = cpc
__first_entry_campaign_id = CAMP-0001
__first_entry_term = obuchenie_partnerov
```

Зачем это нужно:

```
человек мог сначала прийти с рекламы,
потом вернуться из Telegram,
потом нажать ссылку из рассылки.

__entry_* покажет последний вход.
__first_entry_* сохранит первый источник.
```

---

3. Бизнес-слой: `demand_*`

`demand_*` — это удобные поля для фильтрации, сегментации и просмотра лида.

Они короче и понятнее для сценаристов / менеджеров.

Атрибут	Что хранит
`demand_hypothesis_id`	Гипотеза
`demand_segment`	Сегмент
`demand_offer_id`	Оффер
`demand_landing_id`	Лендинг
`demand_entry_point_id`	Точка входа
`demand_campaign_id`	Кампания
`demand_creative_id`	Креатив
`demand_funnel_id`	Воронка
`demand_variant`	A/B-вариант
`demand_entry_source`	Источник
`demand_entry_medium`	Medium
`demand_entry_ref`	Ref
`demand_entry_term`	Ключевая фраза
`demand_entry_route`	Route

Префикс можно поменять:

```
businessPrefix: "growth"
```

Тогда поля будут:

```
growth_hypothesis_id
growth_offer_id
growth_campaign_id
...
```

---

4. Legacy-поля: `last_entry_*`

Для совместимости со старыми сценариями компонент может сохранить:

Атрибут	Что хранит
`last_entry_ref`	Ref
`last_entry_src`	Источник
`last_entry_med`	Medium
`last_entry_cmp`	Campaign ID
`last_entry_route`	Route

Этим управляет параметр:

```
saveLegacyAliases: true
```

Если проект новый и legacy не нужен, можно отключить:

```
saveLegacyAliases: false
```

---

5. Compact summary

Компонент дополнительно сохраняет короткую JSON-сводку:

```
__last_deeplink_entry_summary
```

Пример:

```
{
  "action": "diag",
  "route": "demand_voice_diagnostic",
  "h": "HYP-0001",
  "s": "S03",
  "o": "OFF-0001",
  "c": "CAMP-0001",
  "l": "LP-0001",
  "e": "EP-0001",
  "k": "CR-0001",
  "f": "FUN-0001",
  "x": "yandex",
  "m": "cpc",
  "r": "landing",
```

```
  "t": "obuchenie_partnerov",
  "v": "A"
}
```

Это удобно для:

```
быстрой отладки
проверки входа
просмотра лида
AI-анализа
support notification
```

Telegram-уведомление

Компонент может отправить уведомление в Telegram, если включить:

```
notify: {
  enabled: true,
  tokenAttr: "SUPPORT_TELEGRAM_BOT_TOKEN",
  chatIdAttr: "SUPPORT_TELEGRAM_CHAT_ID",
  title: "□□Новый вход Demand Lab"
}
```

Что нужно настроить

В боте должны быть атрибуты:

```
SUPPORT_TELEGRAM_BOT_TOKEN
SUPPORT_TELEGRAM_CHAT_ID
```

И должен быть доступен плагин:

```
Common.Notifications.Telegram
```

Что попадает в уведомление

```
имя лида
messenger id
lead id
messenger
route
action
hypothesis
segment
offer
campaign
landing
entry point
creative
funnel
variant
source
medium
term
ref
```

Важно

Сейчас шаблон уведомления встроен в компонент.

Можно менять только заголовок:

```
title: "□□Новый вход Demand Lab"
```

Полный шаблон сообщения можно будет параметризовать позже, если это реально понадобится.

Как использовать route в MenuBuilder

`DeepLinks` сам не запускает следующий сценарий.

Он только сохраняет route.

Дальше в MenuBuilder делаются условия.

Диагностика

```
return lead.getAttr("__entry_route") === "demand_voice_diagnostic"
```

Консультант / база знаний

```
return lead.getAttr("__entry_route") === "knowledge_consultant"
```

Передача в продажи

```
return lead.getAttr("__entry_route") === "sales_handoff"
```

Показ оффера

```
return lead.getAttr("__entry_route") === "offer_intro"
```

Fallback

```
return lead.getAttr("__entry_route") === "fallback"
```

Fallback-сообщение

Fallback лучше держать в сценарии, а не внутри `DeepLinks`.

Пример:

```
const { sendFormattedMessage } = require("Common.Helpers.SendFormattedMessage")

sendFormattedMessage(`
*Привет.*

Я помогу быстро понять, есть ли у вас задача, где Metabot / Metarex может дать реальное
преимущество.

Формат простой:
вы рассказываете ситуацию голосом, а я раскладываю её по карте:

• какая проблема сейчас болит
• кого нужно довести до результата
• где рвётся путь
• что уже пробовали
• есть ли смысл обсуждать пилот

Если окажется, что это не наш случай — я так и скажу.

Чтобы начать, нажмите *«Пройти диагностику»*.
`, "Markdown")
```

Почему fallback не внутри плагина:

```
в каждом проекте будет разный текст
разный tone of voice
разная воронка
разный CTA
```

`DeepLinks` должен отвечать за вход и route, а не за UX-сообщения проекта.

Связь с Demand Lab Sheet

ID из deeplink должны совпадать с операционным файлом Demand Lab.

```
08_Hypotheses        → HYP-0001
10_Offers            → OFF-0001
11_Creatives         → CR-0001
12_Landings          → LP-0001
13_Campaigns         → CAMP-0001
14_Funnels           → FUN-0001
```

Тогда вся цепочка становится связанной:

```
Demand Lab Sheet
↓
Landing / Entry Point
↓
Deeplink params
↓
DeepLinks.handle()
↓
Lead attrs
↓
Bot route
↓
Diagnostic
↓
Sales summary
↓
Weekly report
```

Проверка и отладка

Если deeplink не работает, проверять по слоям.

1. Сырые параметры

Проверить:

```
sys_last_script_request_params
```

и внутри:

```
deeplink
```

Если там пусто — проблема не в `DeepLinks`, а в том, как deeplink дошёл до сценария.

---

2. Payload

Проверить:

```
__entry_payload_json
```

Там должны быть исходные параметры ссылки.

---

3. Summary

Проверить:

```
__last_deeplink_entry_summary
```

Если summary пустой, значит компонент не выполнился или не получил raw params.

---

4. Route

Проверить:

```
__entry_route
```

Если route не тот:

```
проверить a / action
проверить f / funnel
проверить rt / route
проверить routes.byAction
проверить routes.byFunnel
проверить defaultDemandRoute
проверить fallbackRoute
```

---

5. Business attrs

Проверить:

```
demand_hypothesis_id
demand_offer_id
demand_campaign_id
demand_funnel_id
demand_entry_source
demand_entry_medium
```

Если они пустые, проверить `aliases`.

---

6. Telegram notification

Проверить:

```
notify.enabled = true
SUPPORT_TELEGRAM_BOT_TOKEN
SUPPORT_TELEGRAM_CHAT_ID
Common.Notifications.Telegram
```

Если уведомление не пришло, но attrs сохранились — проблема только в notification layer.

---

7. MenuBuilder conditions

Проверить, что условия смотрят именно на:

```
lead.getAttr("__entry_route")
```

а не на старые или случайные поля.

Частые ошибки

1. Перепутали action и route

Неправильно:

```
?a=demand_voice_diagnostic
```

Лучше:

```
?a=diag
```

или явно:

```
?rt=demand_voice_diagnostic
```

2. Передали `utm_campaign`, но не настроили aliases

Если в `aliases.campaignId` нет `utm_campaign`, компонент не поймёт campaign.

3. Используют `rt` слишком часто

`rt` удобно, но если все ссылки напрямую задают route, теряется управляемость через `action` и `funnel`.

Обычный рекламный вход лучше делать так:

```
?a=diag&f=FUN-0001&h=HYP-0001&c=CAMP-0001
```

А `rt` использовать для специальных случаев.

4. Не сохраняют `entry.route`

Если после `handle()` не сделать:

```
lead.setAttr("__entry_route", entry.route)
```

MenuBuilder может не увидеть route.

Если сам плагин уже сохраняет `__entry_route`, эта строка всё равно не вредна: она явно показывает сценаристу, что route — главный результат обработки.

5. Нет единого ID-справочника

Если в Excel кампания называется `Campaign 1`, а в ссылке `CAMP-0001`, потом будет сложно связать данные.

Нужны единые ID:

```
HYP-0001
OFF-0001
LP-0001
CAMP-0001
CR-0001
FUN-0001
```

Кратко

```
__entry_* — последний вход
__first_entry_* — первый вход
demand_* — бизнес-поля для фильтрации
last_entry_* — legacy-совместимость
__last_deeplink_entry_summary — compact JSON для отладки

DeepLinks вычисляет route.
MenuBuilder использует route.
Сценарии уже сами показывают сообщения, запускают диагностику, VoiceInput, LLMQuery и продажи.
```

Deep Links — Appendix

Чеклист внедрения и будущее развитие

Туда положить:

```
1. Чеклист установки в нового бота
2. Минимальный боевой пример для Demand Lab
3. Какие bot attrs нужны
4. Какие route conditions создать
5. Какие ID должны совпадать с Excel
6. Что проверять перед запуском рекламы
7. Будущее развитие: demand_entry_events, route config JSON, notification templates
```

То есть не “next” как большая глава, а **финальная служебная часть для разработчика**, чтобы он мог быстро внедрить и проверить.

Минимальный appendix можно сделать таким:

```
# Appendix. Чеклист внедрения DeepLinks


1. Создать / подключить плагин:
Common.Platform.DeepLinks
```

```
2. В стартовом deeplink-сценарии вызвать:
DeepLinks.handle({ lead, ...config })

3. Передать боевую конфигурацию:
aliases
actionAliases
routes
notify
businessPrefix

4. Проверить bot attrs для Telegram-уведомлений:
SUPPORT_TELEGRAM_BOT_TOKEN
SUPPORT_TELEGRAM_CHAT_ID

5. Создать условия MenuBuilder:
__entry_route === demand_voice_diagnostic
__entry_route === knowledge_consultant
__entry_route === sales_handoff
__entry_route === offer_intro
__entry_route === fallback

6. Проверить тестовую ссылку:
?a=diag&h=HYP-0001&o=OFF-0001&c=CAMP-0001&l=LP-0001&f=FUN-
0001&x=yandex&m=cpc&t=obuchenie_partnerov&k=CR-0001&v=A

7. Проверить, что записались:
__entry_payload_json
__last_deeplink_entry_summary
__entry_route
demand_hypothesis_id
demand_campaign_id
demand_funnel_id

8. Проверить, что route ушёл в нужный сценарий.

9. Проверить, что уведомление пришло в Telegram.

10. Только после этого запускать рекламный трафик.
```

И короткий roadmap:

Будущее развитие:

1. demand_entry_events — таблица истории всех входов
2. route config из bot attr JSON
3. шаблоны Telegram notification
4. валидация ID против справочников Demand Lab
5. экспорт entry events в аналитику / WayLogger
6. связь с weekly reports и рекламными метриками