Common.Observability.* — Инструменты наблюдаемости

Observability — пакет системных инструментов для наблюдаемости, диагностики и эскалации событий в Metabot. Предназначен для фиксации, анализа и понимания поведения системы без влияния на бизнес-логику.

- Common.Observability.Tracer — Универсальный трассировщик событий
- Common.Observability.Incident — Централизованный обработчик инцидентов

Common.Observability.Tracer — Универсальный трассировщик событий

Автор: Art Yg

Tracer предназначен для записи любых диагностических событий в таблицы базы данных:

- ошибок
- проверок
- ветвлений логики
- внутренних состояний
- отладочной информации

Tracer **не знает**:

- что такое сессия
- что такое скрипт, команда или триггер
- какие поля считаются «правильными»

Он записывает **ровно те данные, которые ему передали**.

Основные принципы

- **Stateless** — не хранит состояние
- **Schema-agnostic** — не требует фиксированной схемы
- **Zero mandatory fields** — нет обязательных полей
- **Opt-in** — работает только если включён
- **Side-effect only** — не влияет на выполнение кода

Tracer можно удалить из проекта — бизнес-логика продолжит работать.

Минимальные требования

Для начала работы достаточно:

1. Создать любую таблицу в БД (даже без полей, кроме `id`)
2. Добавить атрибут бота `TRACER_CONFIG`
3. Вызвать `trace()`

Рекомендуемое, но **не обязательное** поле таблицы:

- `created_at` с автозаполнением (`NOW`)

Tracer **не управляет временем**. Если поле есть — БД заполнит его автоматически. Если нет — запись всё равно создаётся.

Конфигурация

Tracer настраивается через один JSON в атрибутах бота: `TRACER_CONFIG`.

```
{
  "navigation": {
    "enabled": true,
    "table": "nav_trace"
  },
  "ai": {
    "enabled": false,
    "table": "ai_trace"
  }
}
```

- каждый tracer имеет имя (`navigation`, `ai`, `api` и т.д.)
- у каждого tracera своя таблица (или общая)
- выключенный tracer ничего не пишет

Использование

```
const Tracer = require("Common.Observability.Tracer");

Tracer.trace("navigation", {
  category: "NAVIGATION",
  component: "Actor",
  action: "hasAchievement",
  level: "OK",
  payload: {
    actor_id: 42,
    achievement: "first_step",
    result: true
  }
});
```

Если tracer выключен — метод молча завершится.

Методы Tracer

Tracer предоставляет несколько эквивалентных методов:

- `trace(name, data)` — базовый метод записи
- `log(name, data)` — алиас для читаемости
- `info(name, data)` — добавляет `level: "INFO"`
- `error(name, data)` — добавляет `level: "ERROR"`

Все методы:

- не выбрасывают ошибок
- не изменяют переданные данные
- не влияют на бизнес-логику

Данные события

Tracer принимает **любой объект**.

Все поля:

- опциональны

- именуются произвольно
- записываются «как есть»

Рекомендуемые (но не обязательные):

- `category` — область (NAVIGATION, AI, API)
- `component` — компонент
- `action` — действие
- `source` — источник (system, user, webhook)
- `level` — уровень ошибки
- `payload` — любые данные (тип поля TEXTAREA)

Если таблица не содержит поле — БД вернёт ошибку. В таком случае используйте `payload`.

Работа со временем

Tracer:

- не добавляет timestamp
- не требует поля времени
- позволяет передать своё время

```
{
  event_time: "2026-01-16T12:00:00Z"
}
```

или

```
{
  created_at: "2026-01-16T12:00:00Z"
}
```

Когда использовать

- метод возвращает `true / false`, но нужна диагностика
- не хочется усложнять ответы ошибками
- важно понять, **почему** логика не сработала
- нужна отладка без влияния на сценарии

Когда не использовать

- как бизнес-лог
- как аудит-лог
- как аналитику или метрики

Итог

`Common.Observability.Tracer` — простой и ненавязчивый способ видеть, что происходит внутри системы.

Никакой магии. Никаких обязательств. Никакой боли.

Версия 1.1 — Интеграция с Incident

В версии **1.1** Tracer получил **опциональную интеграцию с системой инцидентов**.

Tracer по-прежнему:

- не знает, что такое уведомления;
- не содержит логики доставки;
- не требует дополнительных обязательных полей.

Интеграция включается **исключительно через конфигурацию**.

Что добавлено

Tracer теперь может:

- автоматически инициировать **Incident** при записи события уровня `ERROR`;

- делать это **без изменения существующих вызовов** `Tracer.error()` и `Tracer.trace()`;
- работать с инцидентами как с **побочным эффектом**, не влияя на основной код.

Если интеграция не настроена — Tracer ведёт себя **точно так же, как в версии 1.0**.

Расширенная конфигурация

В `TRACER_CONFIG` можно указать блок `incident` для любого tracera:

```
{
  "navigation": {
    "enabled": true,
    "table": "nav_trace",
    "incident": {
      "enabled": true,
      "type": "navigation_failed",
      "severity": "error"
    }
  }
}
```

Поведение:

- `incident.enabled = true` — включает обработку инцидентов
- `type` — логический тип инцидента (используется Incident)
- `severity` — уровень инцидента (опционально)

Инцидент инициируется **только если**:

- tracer включён;
- событие имеет `level = ERROR` (регистр не важен).

Архитектурные гарантии

Интеграция с Incident:

- **никогда не ломает выполнение сценария**;
- **не выбрасывает исключений**;
- **не требует изменений в сценариях**;
- полностью **отключается конфигурацией**.

Tracer остаётся:

- stateless,
- schema-agnostic,
- opt-in,
- side-effect only.

Использование (без изменений)

```
Tracer.error("navigation", {
  component: "Reflection",
  action: "build",
  payload: {
    reason: "profile_missing"
  }
});
```

Если Incident включён — будет создан инцидент. Если нет — только запись в таблицу.

Итог версии 1.1

Версия **1.1** добавляет Tracer’у способность **сигналить о сбоях**, не превращая его в логгер, нотификатор или бизнес-модуль.

Tracer по-прежнему ничего не «решает». Он просто сообщает — системе, а не человеку.

Common.Observability.Incident — Централизованный обработчик инцидентов

Common.Observability.Incident

Incident — централизованный обработчик инцидентов в Metabot. Предназначен для регистрации и уведомления о сбоях, ошибках и критических состояниях системы.

Класс **не содержит бизнес-логики** и не влияет на выполнение сценариев. Он используется как реакция на события (например, из Tracer) или вызывается напрямую из сценариев.

Назначение

Incident решает одну задачу: **превратить техническое событие в уведомление для команды.**

Типичные случаи:

- сбой профилирования;
- ошибка отражения / навигации;
- неконсистентное состояние данных;
- критические ошибки AI / runtime.

Уведомления

В текущей реализации Incident использует Telegram через плагин:

```
Common.Notifications.Telegram
```

Для работы требуются **два атрибута бота**:

- `SUPPORT_TELEGRAM_BOT_TOKEN` — токен Telegram-бота
- `SUPPORT_TELEGRAM_CHAT_ID` — ID группы или канала для уведомлений

Incident сам не хранит токены и не управляет доступами.

Шаблоны сообщений

Тексты уведомлений настраиваются через атрибут бота `INCIDENT_TEMPLATES` (JSON).

Пример:

```
{
  "profiling_failed": {
    "ru": {
      "title": "□□ORION · Сбой профилирования",
      "body": [
        "Профиль не был корректно сформирован.",
        "",
        "Lead ID: {{lead_id}}",
        "{{error}}"
      ]
    }
  }
}
```

Поддерживаются:

- разные типы инцидентов;
- несколько языков;
- плейсхолдеры `{{variable}}`.

Использование

Incident может вызываться:

- напрямую из сценариев;
- автоматически из `Common.Observability.Tracer` (при включённой конфигурации).

Это позволяет централизовать обработку ошибок **без дублирования кода уведомлений**.

Incident — это точка ответственности за инциденты, а не ещё один логгер или бизнес-модуль.